Conception et évaluation d'un protocole de reprise d'applications parallèles dans une fédération de grappes de calculateurs

Sébastien Monnet (Sebastien.Monnet@irisa.fr)

22 juin 2003

Table des matières

Table des figures

Liste des Algorithmes

Résumé

Ce document présente le compte-rendu d'une étude sur la reprise d'applications parallèles dans les fédérations de grappes de calculateurs. Les protocoles de l'état de l'art ne passent pas à l'échelle. Ce doccument décrit un protocole hiérarchique de points de reprise / recouvrement arrière qui combine une technique de points de reprise coordonnés au sein d'une grappe et une technique de points de reprise induits par les communications entre les grappes. En premier lieu, il présente un aperçu des techniques de reprises d'applications parallèles existantes ainsi qu'une discussion sur ces dernières. Ensuite, une réflexion sur la prise en compte de l'architecture particulière des fédérations de grappes de calculateurs est menée. Cette réflexion aboutie à la proposition d'un protocole de tolérance aux défaillances adapté. Ce protocole est présenté et analysé au travers d'exemples. Une étude des performances du protocole proposé est menée à l'aide d'un simulateur à évènements discrets.

Remerciements

Je remercie mes encadrants Christine Morin et Ramamurphy Badrinath pour leur conseils et le temps qu'ils m'ont consacré. Je remercie également Mr Thierry Priol de m'avoir permis d'entrer dans son équipe, ainsi que tous les membres du projet Paris pour leur accueil chaleureux.

Chapitre 1

Introduction

Les grappes de calculateurs offrent une alternative aux machines parallèles pour l'exécution d'applications parallèles nécessitant de la haute performance. Les progrès technologiques concernant les communications permettent d'utiliser des réseaux à très haut débit et faible latence ce qui permet, dans certains cas, aux grappes de concurrencer les machines parallèles. Les grappes de calculateurs sont non seulement moins chères mais elles sont également évolutives : il est possible d'ajouter du matériel sans avoir à changer toutes les machines. Cependant, la multiplication de matériel standard (par exemple des PC) augmente la probabilité d'apparition de défaillances. Les grappes de calculateurs étant souvent utilisées pour l'exécution d'applications de longue durée, il est intéressant de mettre en place des mécanismes de tolérance aux fautes. En effet, la durée d'exécution de certaines applications peut être supérieure au temps moyen entre deux défaillances (MTBF) d'une grappe.
Les problèmes rencontrés lors de la mise en œuvre de techniques de reprise d'applications parallèles dans des grappes de calculateurs sont nombreux. Cela fait maintenant plus d'une vingtaine d'années que des chercheurs proposent des solutions. De ce fait, un grand nombre de ces mécanismes sont présentés dans la littérature.
De nouveaux types d'architecture voient le jour : les fédérations de grappes de calculateurs. Il s'agit de fédérer plusieurs grappes qui peuvent être géographiquement distribuées, reliées par des réseaux de type LAN ou WAN. Par exemple, une application numérique s'exécute sur une grappe de calculateurs, produisant une trace qui est analysée sur une deuxième grappe. Dans ce type d'architecture, le nombre de nœuds est très important (le MTBF du système complet est inférieur au minimum des MTBF de chaque grappe). La mise en place de mécanismes de tolérance aux fautes dans les fédérations de grappes de calculateurs devient essentielle. Cette architecture présente des particularités (nombre de nœuds, caractéristiques réseau) qu'il est nécessaire de prendre en compte. Notre travail de stage a porté sur la conception d'un protocole de recouvrement arrière adapté à des applications parallèles exécutée dans une fédération de grappes. Nous proposons un protocole hybride de reprise d'applications parallèles combinant synchronisation au sein des grappes, et non-coordination entre les grappes. L'évaluation de ce protocole est faite par simulation avec des traces synthétiques.
La suite de ce document se décompose comme suit. Le second chapitre présente un état de l'art des techniques de sauvegarde point de reprise et de retour arrière dans les systèmes distribués. Le troisième présente les principes de conception d'un protocole hiérarchique de reprise d'applications paralléles dans une fédération de grappes de calculateurs, ainsi que le protocole proposé lui-même. Le chapitre 4 présente le simulateur à évènements discrets que nous avons réalisé pour l'évaluation

du protocole proposé ainsi que les résultats de cette évaluation. Dans le chapitre 5, nous présentons le bilan de nos travaux ainsi que les perspectives offertes.

Chapitre 2

Principes et techniques de tolérance aux fautes

2.1 Introduction

Ce chapitre présente un état de l'art des diférentes techniques de points de reprise pour les applications parallèles. Le principe fondamental des techniques de points de reprise est d'effectuer des sauvegardes régulières de l'état des applications et de relancer ces dernières à partir de ces points de sauvegarde, appelés points de reprise *(checkpoints)* par la suite.
Les applications parallèles reposent sur différents paradigmes de programmation : échanges de messages ou mémoire partagée. Dans les grappes de calculateurs, les environnement MPI ou PVM supportent le premier modèle, les systèmes de mémoire partagée distribuée (DSM) le second modèle. Il semble judicieux de proposer au programmeur une abstraction du système distribué en mettant en œuvre des modèles de mémoire partagée distribuée, offrant ainsi l'illusion d'une machine parallèle unique. Ce procédé permet également l'utilisation d'une grappe par des applications déjà existantes prévues pour des multiprocesseurs. Le fonctionnement des systèmes à mémoire partagée distribuée, bien que s'appuyant sur des échanges de messages requiert des techniques de tolérance aux défaillances adaptées. Ce chapitre présente donc des mécanismes pour ces deux types de paradigmes. Le pragraphe suivant expose des généralités et apporte quelques définitions. le troisième présente les techniques de points de reprise coordonnés, la quatrième celles de points de reprise indépendants. Enfin, le cinquième paragraphe consiste en une discussion autour des différentes solutions proposées et une ouverture sur la tolérance aux défaillances dans les fédérations de grappes de calculateurs.

2.2 Généralités

La durée d'exécution des applications pouvant être longue et la probabilité de défaillances élevée, il est intéressant de pourvoir reprendre en cas de défaillance l'exécution à partir d'un point de reprise de l'application et non du début, notamment lorsque le temps moyen entre deux défaillances

est inférieur au temps d'exécution de l'application. Pour ce faire, il est nécessaire de sauvegarder l'état des processus régulièrement. Cependant, les processus communiquant entre eux, cela crée des dépendances. De ce fait, les différents états locaux enregistrés pour chacun des processus ne forment pas, dans le cas général, un état global cohérent.

2.2.1 Modèle de système

Une application parallèle est composée de plusieurs processus s'exécutant en parallèle (sur des sites éventuellement différents) et communiquant entre eux. Cette communication peut s'effectuer soit via des échanges de messages, soit via une mémoire partagée.
Dans le cas où les processus s'exécutent sur des sites différents (ce qui est le cas dans les grappes de calculateurs) le système offrant la mémoire partagée utilise de manière sous-jacente des échanges de messages. Afin de gérer la mémoire partagée, le système doit tenir à jour des structures de données particulières (afin de retenir quel est le site de référence pour une page mémoire particulière par exemple). Les mécanismes de tolérance aux défaillances doivent tenir compte de ces dernières.
Il existe différents modèles de cohérence pour les DSM : séquentielle ou relâchée. Dans le modèle de cohérence forte, à chaque instant, tous les processus ont la même vision de l'état de la mémoire. Le protocole utilise souvent des mécanismes d'invalidation, c'est à dire qu'avant d'écrire dans une variable ou une page de mémoire partagée (selon le grain de partage choisi), un processus invalide toutes les copies détenues par les autres processus. Le modèle de cohérence relachée, également appelé modèle de cohérence "à la libération" garantit que tous les processus ont la même vision d'une variable à sa libération. Ceci oblige le programmeur à acquérir et relacher des verrous avant d'effectuer des opérations en mémoire, mais apporte généralement des gains de performances. Les différences entre ces deux modèles sont à prendre en compte lors de la mise en place de techniques de tolérance aux fautes.
Un point de reprise pour un processus est un ensemble d'éléments permettant de relancer ce processus (l'état de la mémoire par exemple). Il peut être établi de manière incrémentale: seules les données modifiées depuis le dernier point de reprise sont sauvegardées. Certains systèmes permettent la duplication de processus *(fork)*, ce qui autorise la création du point de reprise en arrière plan (sans bloquer le processus). Pour une application parallèle composée de plusieurs processus, un point de reprise est composé d'un ensemble de points de reprise des processus de l'application. La partie suivante décrit plus en détail cet ensemble de points de reprise.
En cas de défaillance, les états sauvegardés doivent pouvoir être accessibles. Les points de reprise sont donc stockés dans des espaces de stockage stables. Un espace de stockage est dit "stable" si il possède les propriétés d'accessibilité, d'inaltérabilité et d'atomicité. C'est-à-dire, qu'un tel espace de stockage doit toujours être accessible et que les données qu'il contient doivent être cohérentes. Afin d'assurer la cohérence des données, les écritures doivent être atomiques (i.e. elles sont soit effectuées dans leur intégralité, soit abandonnées), de plus une fois écrites, les données ne doivent pas être altérées. Dans la pratique il peut s'agir d'un disque RAID, de duplication des données sur différentes machines de la grappe (sur disque ou en mémoire),... L'atomicité des écritures est généralement assurée par des mécanismes de validations.

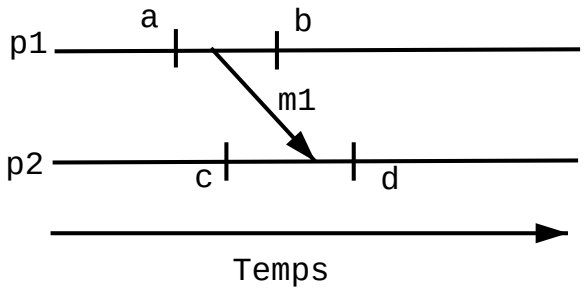

FIG. 2.1 – Dépendance entre deux processus

2.2.2 Dépendance et cohérence

Pour mettre en œuvre de la tolérance aux défaillances au sein d'une application séquentielle constituée d'un seul processus, il est possible d'effectuer des sauvegardes de l'état du processus de temps en temps. En cas de défaillance, le processus est relancé à partir de la dernière sauvegarde.
Dans le cas d'une application parallèle constituée de plusieurs processus, la communication inter-processus entraîne des dépendances empêchant l'utilisation de ce même procédé indépendamment sur chacun des processus.
L. Lamport a défini la relation de causalité entre deux évènements *(happened before)*. Au sein d'un même processus, un évènement en précède un autre si il se produit avant dans le temps. Entre deux processus communicants, l'émission d'un mesage par un processus précède la réception de ce même message par un deuxième processus. La relation de causalité est transitive, c'est-à-dire que si un évènement x précède un évènement y et si y précède un évènement z, alors x précède z. Une exécution répartie peut donc être vue comme un treilli d'évènements.
Sur la figure 2.1, le processus *p1* envoie le message *m1* au processus *p2*. L'évènement "envoi de *m1*" sur *p1* précède (selon la relation de causalité) l'évènement "réception de *m1*" sur *p2*. Si aprés une défaillance *p1* est relancé en *b* et *p2* en *c*, *p1* a émis *m1* alors que *p2* ne l'a pas encore reçu. Dans ce cas de figure, *m1* est appelé un message en transit. Certaines solutions consistent à journaliser les messages reçus afin de rejouer la réception aprés un retour arrière. En revanche, si *p1* est relancé en *a* et *p2* en *d* alors *p1* réémet *m1* et *p2* le reçoit donc 2 fois (de plus, les deux messages reçus par *p2* ne sont pas forcément identiques). Dans ce cas, le premier message *m1* est appelé un message fantôme *(ghost message)*, il a été reçu alors qu'il n'a pas encore été émis. Les échanges de messages entre processus entrainent des dépendances, il est donc nécessaire de bien choisir les moments auquels sauvegarder des points de reprise pour chacun des processus.
Une dépendance entre deux processus est engendrée par un envoi de message dans les modèles à échanges de messages, et par des séquences "écriture/lecture" ou "écriture/écriture" d'une même zone mémoire pour les modèles de mémoire partagée distribuée.
Un ensemble de points de reprise valides est appelé "ligne de récupération". Une ligne de récupération (formée d'un ensemble d'états locaux sauvegardés) correspond à un état global cohérent [7].

2.2.3 Etat global cohérent

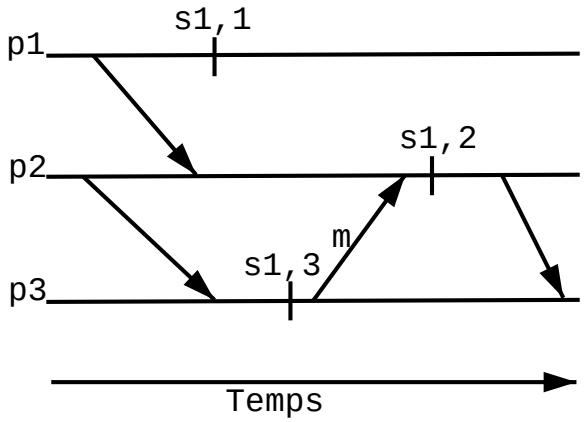

FIG. 2.2 – Recherche d'un état global cohérent

Sur la figure 2.2, nous considérons 3 processus, s(i,j) représente le i^{éme} point de reprise local pris par le processus p_j. Si l'on considère l'état global formé par les $s_{1,j}$, j=1..3, on s'aperçoit que le processus p_2 a reçu le message m (i.e le message m est inclus dans l'histoire de $s_{1,2}$) alors que le processus p_3 ne l'a pas encore envoyé. Il s'agit donc d'un message fantôme rendant l'état global $s_{1,j}$ non cohérent, ne pouvant pas être utilisé pour la reprise après une défaillance. Il est possible de construire des exécutions dans lesquelles aucun ensemble des états locaux sauvegardés ne correspond à un état global cohérent. Dans ce cas, en cas de faute d'un des processus, lors de la reprise, tous les processus de l'application vont devoir revenir en arrière jusqu'au point de départ (dans le pire des cas). Ce phénomène est connu sous le nom d'effet domino. L'effet domino correspond au cas où la seule ligne de récupération valide est le point de départ de l'application.
Le calcul d'un état global cohérent a donné lieu à de nombreuses recherches, une solution élégante a été donnée par K.M Chandy et L. Lamport [7]. Cette solution utilise un système de "marqueurs" envoyés par le site initiateur du calcul de l'état global cohérent puis retransmis par les processus le recevant (après avoir créé un point de reprise local). En supposant les canaux de communication FIFO, les marqueurs permettent aux processus de journaliser les messages reçus des sites n'ayant pas encore transmis de marqueur. Tous les messages envoyés par la suite à ces derniers leur parviendront après le marqueur (les canaux étant supposés FIFO) ceci élimine la possibilité d'avoir des messages fantômes (chaque processus recevant un marqueur de tous les autres). Cette technique, cependant ajoute des messages lors de la création d'un point de reprise, de plus elle nécessite la journalisation de messages reçus, en effet dans l'état restauré, ils ont été envoyés mais non reçus, le processus destinataire doit donc les "rejouer". Le système de tolérance aux fautes décrit dans [36] utilise un procédé semblable à celui-ci. Dans ce dernier, les processus confirment le point de reprise au processus initiateur qui envoie un *"commit"* à tous les processus lorsque le calcul est terminé.

2.2.4 Techniques de points de reprise

Dans les solutions proposées pour les modèles de sauvegarde/reprise, on distingue deux grandes familles de techniques. La difficulté consiste à trouver une ligne de récupération.
La première famille de protocoles, pessimistes, consiste à créer une ligne de récupération à la sauvegarde. Ce type de technique est dit "coordonné". En effet, cela consiste à synchroniser les processus lors de la sauvegarde, évitant ainsi tout effet domino et simplifiant la gestion des données de reprise en local. Cette technique est détaillée dans la troisième partie.
La deuxième famille, optimiste, ne calcule la ligne de récupération que lors de la reprise aprés une défaillance. Les points de reprise sont sauvegardés pour chacun des processus indépendament les uns des autres. Cela évite le surcoût dû à la synchronisation mais rend les techniques de reprise plus complexes. Cette famille de techniques est détaillée dans la quatrième partie.
La grande diversité des solutions proposées s'explique par l'hétérogénéité des systèmes qui vont les exploiter: par exemple, certaines sont conçues pour des systèmes à très grande échelle [5] dans lesquels une synchronisation des processus serait très coûteuse (voir impraticable), d'autres sont adaptées aux grappes de calculateurs utilisant un réseau à très haut débit et faible latence. Les solutions proposées pour les systèmes à mémoire partagée distribuèe prennent en compte les particularités de ces dernières, et diffèrent donc de celles proposée pour les systèmes à échange de messages.
En effet, il existe des solutions permettant de rendre des systèmes à mémoire partagée distribuée fiables ([38][3][9][36][21][30]). Les mécanismes de tolérance aux fautes vont pouvoir exploiter certaines propriétés de ces systèmes. Par exemple, si le modèle de mémoire partagée est à cohérence "à la libération" (*relaxed consistency*) les mécanismes de points de reprise peuvent exploiter les acquisitions/libérations de verrous.

2.2.5 La journalisation

Le fait de sauvegarder des points de reprise afin de permettre un retour arrière n'est pas toujours suffisant. En effet, lors de l'utilisation d'un modèle de points de reprise indépendants, des journaux permettent de rejouer certains évènements pour retrouver un état global cohérent. Ceci de manière à éviter l'effet domino. De plus certaines applications ayant des communications avec le monde extérieur, devront en cas de retour arrière être capable de reproduire ces évènements.
Dans tous les cas, même si l'on oblige les processus à se synchroniser (afin d'établir des points de reprise représentant un état global cohérent), une faute d'un des processus exige, de manière générale, un retour arrière de chacun des processus de l'application (au mieux ceux avec lesquels le processus fautif a communiqué directement ou indirectement depuis le précédent point de reprise sauvegardé). Une journalisation peut alors permettre de limiter le retour arrière au seul processus fautif. Cependant les techniques de journalisation sont plus souvent utilisées dans les systèmes mettant en œuvre des techniques de point de reprise indépendants.
La journalisation des évènements permet au processus fautif de "rejouer" les communications avec les autres processus de l'application, n'imposant ainsi pas un retour arrière à ces derniers. Il faut noter que la journalisation ne concerne pas seulement les communications mais tous les évènements "non déterministes" comme certains appels système, des signaux reçus... Cela suppose que l'on soit capable de détecter ces évenements et de les journaliser. Nous verrons par la suite que cette hypothèse est parfois trés forte.

Il existe deux principales techniques de journalisation: la journalisation “pessimiste” et la journalisation “optimiste”. La première consiste à considérer qu'une panne peut survenir pendant l'écriture du fichier journal, c'est-à-dire que les évènements sont journalisés avant d'être pris en compte par le processus applicatif. Ces évènements peuvent ainsi être reproduit aprés une défaillance. Les modèles de journalisation optimistes considèrent que le risque de défaillance durant l'écriture du journal est faible ce qui permet de ne pas bloquer le processus en attendant que les données journalisées soient sauvegardées sur un espace de stockage stable. En revanche, la reprise après une faute ainsi que le “ramasse miettes”, récupérant l'espace utilisé par les données journalisées obsolètes, sont plus difficile à mettre en œuvre et présentent un surcoût plus important (certains processus non fautifs sont contraints d'effectuer un retour arrière).
Une solution permet d'exploiter les avantages de ces deux techniques: il s'agit de la journalisation causale. Elle se comporte comme la journalisation optimiste dans la majorité des cas, sauf en cas de création de dépendance *happened before* (de L. Lamport) du processus vers un autre dans quel cas c'est la version pessimiste qui est utilisée. En utilisant cette technique, les processus non fautifs ne sont pas contraints à un retour arrière. Dans le pire des cas, le processus fautif doit revenir à son dernier point de reprise. Cependant cela exige d'observer les relations de causalités inter-processus et donc d'émettre des données supplémentaires (comme des mises à jour de graphe de dépendance) qui sont généralement portées par les messages de l'application *(piggybacking)*.

2.3 Point de reprise coordonné

2.3.1 Technique de base

Établir un point de reprise coordonné consiste à synchroniser les processus lors de la sauvegarde des états locaux afin de garantir que l'état global obtenu à l'aide de l'ensemble de ces états locaux soit cohérent. La ligne de récupération est ainsi créée lors de l'établissment du point de reprise. De ce fait, l'application peut être relancée en faisant repartir chacun des processus (ou seulement les fautifs si un système de journalisation est mis en place) au niveau de son dernier point de reprise, aucun autre calcul supplémentaire n'est nécessaire. On remarque qu'il suffit alors de conserver le dernier point de reprise de chaque processus. En effet, tous les états sauvegardés antérieurement deviennent inutiles, ils se situent en amont d'une ligne de récupération. De plus la création d'un point de reprise de ce type permet de purger les journaux (si une journalisation est mise en place) de toutes leurs données (le processus ne reviendra jamais en arrière de la ligne de récupération).
En revanche cette technique présente un inconvénient majeur: elle nécessite la synchronisation de tous les processus, ce qui engendre un surcoût qui peut être élevé, surtout dans des systèmes à très grande échelle où le débit des communications est faible et la latence élevée. De plus certains systèmes sont désireux de tolérer des nœuds volatiles [5], auquel cas, une synchronisation s'avère impossible.

2.3.2 Optimisations de la technique de base

Il existe de nombreuses améliorations possibles pour limiter le surcoût dû à la synchronisation des processus [29] et [11].
En premier lieu il n'est pas nécessaire que l'intégralité des processus se synchronisent. En effet, seuls les processus ayant des dépendances doivent le faire. Ensuite, la quantité de données à sauvegarder peut être grandement diminuée en utilisant une technique de points de reprise incrémentale. Seules les données modifiées depuis le dernier point de reprise sont alors sauvegardées.
Les processus ne sont pas nécessairement bloqués durant la durée de la sauvegarde de l'état global, il est possible sur des systèmes de type Unix d'effectuer un *fork* afin de répliquer le processus et d'effectuer la sauvegarde à partir de la réplique, c'est le procédé qu'utilise [5] et CONDOR,.
[36] utilise les services de GENESIS, un système pour les grappes de PC proposant un modèle de mémoire partagée afin de ne bloquer les processus que si cela est nécessaire (i.e. si leur exécution n'entraîne pas de dépendances, comme l'envoi de messages). En effet, GENESIS propose des services systèmes permettant de bloquer les processus, de les migrer, de les dupliquer, de gérer les données en mémoire, les accés réseaux.
Une autre optimisation consiste à profiter des barrières de synchronisation de l'application pour effectuer les points de reprise. [9] implémenté sur *TreadMarks* qui effectue (entre autres) des points de reprise coordonnés, profite de la synchronisation du système lors du lancement du "ramasse miettes" pour effectuer ce type de point de reprise.
Une solution apportée dans [30] pour les systèmes à mémoire virtuelle partagée exploite la réplication des données inhérente à ces systèmes pour la sauvegarde des points de reprise. Ceci est réalisé en étendant le protocole de cohérence du système. La présence de répliques sur différents sites permet de minimiser les transferts de données, de plus, le fait que les données soient conservés dans la

mémoire de plusieurs nœuds et non sur disque permet des accés rapides aux données de reprise. Une des améliorations évoquée plus haut est également utilisée ici: seules les données ayant été modifiées depuis le dernier point de reprise sont sauvegardées, il s'agit donc d'une méthode incrémentale. De surcroit, les données utiles pour la tolérance aux fautes peuvent être exploitées par le système de mémoire partagée ce qui permet de compenser en partie le surcoût dû aux mécanismes de tolérance aux fautes.

2.4 Point de reprise indépendant

2.4.1 Technique de base

Contrairement au point de reprise coordonné, le point de reprise indépendant ne nécessite pas de synchronisation inter-processus au moment de la sauvegarde des points de reprise. Les processus sauvegardent leur état de temps en temps de manière indépendante. Le surcoût dû à la synchronisation est ainsi évité, de plus chaque processus peut ainsi choisir un moment propice à la sauvegarde de son état (quand il y a peu de données à sauvegarder par exemple). Ceci rend cette technique intéressante lors des exécutions sans fautes. Il faut tout de même pondérer cette dernière affirmation, en effet, les états sauvegardés n'appartenant pas nécessairement à un état global cohérent, le nombre d'états à conserver pour la reprise peut être très important introduisant ainsi un surcoût en terme d'espace de stockage sur chaque nœud. De plus, afin d'éviter des successions de retour arrière dûes à l'effet domino, des mécanismes de journalisation doivent être mis en place, apportant également un surcoût, y compris lors des exécutions sans fautes. Enfin, la reprise après faute est complexifiée, il faut en effet calculer la ligne de recouvrement, c'est-à-dire jusqu'où les processus doivent revenir en arrière afin de redémarrer sur un état cohérent.

2.4.2 Optimisations de la technique de base

Points de reprise induits

Une technique qui peut être vue comme une variante du point de reprise indépendant est le point de reprise induit par les communications. Il s'agit d'effectuer des points de reprise indépendants comme décrit ci-dessus, mais, afin d'éviter l'effet domino, les processus sont contraints de temps en temps à sauvegarder leur état. En effet, si l'on force un processus à sauvegarder son état à chaque réception de message (pouvant engendrer une dépendance) cela élimine l'effet domino au prix de nombreux points de reprise inutiles.
Cependant, des solutions permettent de limiter ce nombre grâce à l'utilisation de données supplémentaires portées par les messages de l'application (des graphes ou parties de graphes de dépendances). Il n'y a toutefois pas de synchronisation, les processus n'ont donc pas globalement connaissance de l'état du système (les graphes de dépendances sont incomplets), et il y a plus de sauvegardes contraintes que nécessaire (propriété de sûreté).

Mise en œuvre dans des DSM à cohérence relâchée

Un des problèmes généré par l'utilisation de la technique de points de reprise indépendants est la quantité de données générées: tandis que dans la technique du point de reprise coordonné un seul point de reprise doit être sauvegardé sur chaque processus, ici, le nombre de point de reprise retenus ainsi que la taille des fichiers journaux peuvent être très importants. Une réponse à ce problème est exposée dans [38]. Cet article expose une technique pour implanter un ramasse miettes efficace dans un système à mémoire partagée à cohérence relachée utilisant un mécanisme de point de reprise indépendant et de la journalisation. Cette technique s'appuie sur deux algorithmes qui permettent d'obtenir une taille de journal (*Lazy Log Trimming* ou LLT) ainsi qu'un nombre de points de reprise utiles à conserver (*Check Garbage Collection* ou CGC) limités (borne prouvée dans le document). Le "ramasse miette" mis en place dans cet article est distribué et n'effectue pas de synchronisation, cela reviendrait en effet à utiliser une technique de point de reprise coordonné. Aucun envoi de message n'est ajouté, et seules les données du protocole de mémoire partagée sont utilisées. Ce mécanisme s'appuie sur les propriétés du système de mémoire partagée qui associe à chaque page un nœud de référence. C'est la notion d'intervalle de temps associée avec des vecteurs d'horloge qui permet aux algorithmes de déterminer quelles données peuvent être supprimées.

[9] décrit une implémentation de la tolérance aux défaillances dans *TreadMarks* (système à modèle de mémoire partagée à cohérence relâchée). Dans la partie concernant les solutions proposées dans le cadre des techniques de points de reprise synchronisés, cette solution a déjà été abordée. En effet [9] propose une solution employant une technique de points de reprise indépendants et de la journalisation en s'appuyant sur des journaux mis en place par *TreadMarks* mais également des points de reprise synchronisés lors de l'exécution du "ramasse miette". Ce système s'appuie sur le fait que *TreadMarks* conserve presque toutes les données de journalisation nécessaires dans ses structures de données. Le mécanisme ajoute des paires de vecteurs d'horloges sur les sites émetteurs et récepteurs lors des communications. Aucun message supplémentaire n'est transmis. Lors de la reprise, le processus fautif est relancé à partir de son dernier point de reprise, et les données présentes dans les journaux de *TreadMarks* associées aux vecteurs d'horloge ajoutés permettent de rejouer les évènements survenus entre temps, sans forcer les autres processus à revenir en arrière.

Tolérance aux fautes à très grande échelle

Le système MPICH-V ([5]) utilise une version de MPI (système à échange de messages) afin de permettre l'exécution d'applications sur des systèmes distribués à l'échelle d'Internet. Ceci implique la nécessité de prendre en compte la volatilité des nœuds et rend une synchronisation entre tous les processus difficilement réalisable. Aussi, MPICH-V utilise un système de points de reprise indépendants et une journalisation pessimiste. MPICH-V implémente toutes les fonctionnalités de MPICH (implémentation de MPI). Son architecture est composée de quatre principaux types de nœuds: le *manager*, les *serveurs de mémoires de canaux*, les *serveurs de points de reprise*, et les nœuds de calcul. Le *manager* est responsable de la gestion des tâches, de l'ordonnancement, il gère à l'aide de registres les associations entre les nœuds et les *mémoires de canaux* et *serveurs de points de reprise*, détecte également les défaillances par la réception ou non de messages "Je suis en vie" périodiques. Les *serveurs de mémoires de canaux* reçoivent tous les messages, et sont associés aux destinataires,

ils permettent ainsi la reprise (en collaboration avec les *serveur de points de reprise*). Les *serveurs de points de reprise* stockent les images correpondant aux points de reprise et s'accordent avec les *serveurs de mémoires de canaux* afin que la reprise soit cohérente. Les performances d'un tel système sont difficiles à évaluer, elles dépendent beaucoup du nombre de *serveurs de mémoires de cannaux* et de *serveurs de points de reprise* par nœud. En effet, ces derniers constituent un goulet d'étranglement. L'article apporte la preuve que MPICH-V est tolérant aux défaillances, cependant, il faut noté que cela concerne la défaillance des nœuds de calcul, les *serveurs de mémoires de canaux* par exemple sont supposés fiables. Il existe d'autres implémentations de ce type, par exemple MPI-FT est un système proche de MPICH-V, la principale différence étant que MPI-FT est construit sur LAM-MPI (une autre implémentation de MPI) au lieu de MPICH.

2.5 Discussion

Lors de la mise en œuvre de techniques de tolérance au fautes, des études comme [30][36][38] et [9] ont montré qu'il était possible d'utiliser des données ou services ([36]) apportés par le système de mémoire partagée distribuée pour effectuer des sauvegardes d'états ([30]), de la journalisation ([9]) ou faciliter le travail du "ramasse miettes" ([38]) dans le cas du point de reprise indépendant avec journalisation. De plus les données apportées par les procédés de fiabilité peuvent être utilisées par le système de mémoire partagée, l'étude [30] montre même qu'il y a des cas dans lesquels le système est plus performant avec que sans techniques de tolérances aux fautes. Les DSM sont donc bien adaptées à la mise en place de techniques de tolérance aux fautes.
Si l'on compare la technique des points de reprise coordonnés et celle des points de reprise indépendants, la première peut être vue comme une méthode pessimiste dans le sens où elle engendre un surcoût non négligeable y compris lors d'une exécution sans fautes mais permet des reprises efficaces et relativement simples à mettre en œuvre. La deuxième quant à elle est une méthode optimiste, en effet lors d'exécutions sans fautes elle apporte un surcoût relativement faible mais la reprise après une défaillance est très complexe et en général moins efficace que dans le premier cas. Le surcoût de la synchronisation étant directement lié aux performances des communications, les techniques de points de reprise indépendants semblent particulièrement adaptées aux systèmes à grande échelle.
Toutefois les deux techniques de points de reprise peuvent être combinées, par exemple, [9] propose une solution centrée autour de *TreadMarks* dans laquelle les deux types de points de reprise coexistent, ils ne sont donc pas incompatibles. Le système décrit peut être considéré comme utilisant une technique de points de reprise indépendants, profitant de la journalisation présente dans *TreadMarks* pour éviter les inconvénients de ce type de technique. Il effectue également des points de reprise synchronisés en profitant de la synchronisation utilisée par *TreadMark* à des fins de "ramasse miettes". Ce dernier point, en plus de permettre de tolérer plusieurs fautes autorise la purge les fichier journaux et des points de reprise devenus obsolètes.
Il existe également des architectures permettant de mettre en œuvre les deux familles de techniques de points de reprise. C'est le cas de Starfish [1] qui est un environement tolérant aux fautes (lui-même) et permettant de mettre en place des techniques de points de reprise pour les applications. Starfish est un système modulaire dans lequel il est possible d'utiliser soit des points de reprises coordonnés soit des points de reprise indépendants (en choisisant le module de tolérance aux fautes adéquat). Ce type d'architecture peut ainsi permettre de comparer différentes techniques de points de reprise.
La tolérance aux fautes dans des fédérations de grappes hétérogènes nécessite une connaissance de

ces différentes techniques, il peut en effet exister certaines grappes utilisant des systèmes à modèle de mémoire partagée, et d'autre utilisant une interface de type MPI. De plus on doit pouvoir utiliser des techniques de points de reprise différentes, ce qui est à priori de toute façon impliqué par la différence des systèmes sous-jacents. On peut également souhaiter utiliser certaines techniques localement (au niveau d'une grappe) et d'autres pour relier les différents sous-systèmes ainsi créés. Par exemple, au sein d'une grappe utilisant un réseau comme MYRINET des techniques de points de reprise coordonnés sont envisageables ; alors que des synchronisations de processus répartis sur plusieurs grappes qui peuvent être éloignées géographiquement n'est pas souhaitable. La communication inter-grappes peut être effectuée par échange de messages et une technique de points de reprise indépendants utilisant des fichiers journaux (à la manière de MPICH-V par exemple) peut être mise en place. L'utilisation de plusieurs techniques différentes dans un même système (une fédération de grappes par exemple) peut s'avérer intéressant, permettant de bénéficier ainsi de mécanismes spécifiques à chaque niveau afin de limiter le surcoût engendré. En revanche, cette hétérogénéïté implique un niveau supérieur de complexité.

Chapitre 3

Conception d'un protocole de reprise d'applications parallèles hiérarchique

3.1 Objectifs

Le but de ce stage est d'étudier la tolérance aux défaillances pour les applications parallèles s'exécutant sur des fédérations de grappes. Ce type d'architecture peut être employé en particulier pour des applications de type "couplage de code". Typiquement, une grappe peut servir à l'exécution d'une simulation dont la trace générée va être utilisée par une application d'analyse ou une autre simulation sur une deuxième grappe. Les fédérations de grappes sont des systèmes à grande échelle, le temp moyen entre deux défaillances (MTBF) est inférieur au plus petit des MTBF des grappes fédérées. Afin de permettre l'exécution jusqu'à son terme d'une application parallèle dans un tel système, il est essentiel de mettre en œuvre des mécanismes de tolérance aux fautes. Les mécanismes de tolérance aux fautes doivent tenir compte de l'architecture particulière que représente un tel système, ceci afin d'obtenir un surcoût le plus faible posible. Il faut prendre en considération un certain nombre de paramètres: le nombre de nœuds dans ce type d'architecture peut être très important. Les différences de latence et bande passante entre deux nœuds peuvent être grandes selon que ces nœuds appartiennent ou pas à la même grappe. Ceci implique que le surcoût dû à une synchronisation peut être très élevé, de plus le non-déterminisme au sein d'un tel système est important.
Ce chapitre présente le protocole proposé. Les modèles et hypothèses considérés sont exposés. La suite de ce chapitre présente la conception ainsi qu'une description du protocole proposé. Enfin quelques propriètés sont donnée ainsi que des exemples.

3.2 Modèle de système considéré

3.2.1 Architecture

L'architecture considérée est une fédération de grappes de calculateurs homogène.

Grappe de calculateurs Une grappe de calculateurs est un ensemble de machines indépendantes (PC, station de travail) inter-connectées par un réseau de type SAN (Les réseaux de type SAN, *System Area Network* ont généralement des latences faibles et un débit élevé. Nous considérons ici que les machines constituant une grappe ne sont pas dédiées à un utilisateur donné et sont utilisées à des fins de calcul. Ces machines sont généralement situées dans une même pièce pour des raisons de longueur des liaisons réseaux.

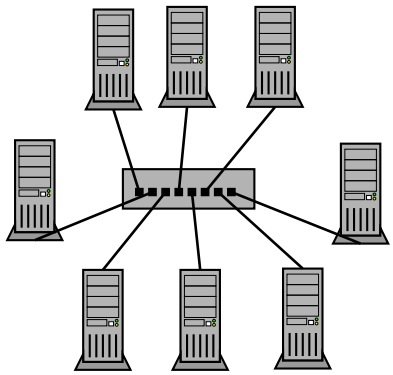

FIG. 3.1 – Une grappe de calculateurs homogène

Fédération de grappes de calculateurs Une fédération de grappes de calculateurs est un ensemble de grappes de calculateurs (comme définies ci-dessus) inter-connectées par un réseau de type WAN (*World Area Network*) ou LAN (*Local Area Network*). Ces liens inter-grappes sont représentés sur la figure 3.2 par des pointillés. Un réseau de type WAN possède une latence plus élevée qu'un réseau de type SAN, et un débit plus faible. Un réseau inter-grappe peut être un réseau à haut débit comme par exemple le réseau VTHD (Vraiment Trés Haut Débit), Internet, ou un réseau d'entreprise de type Ethernet 100 Mb.

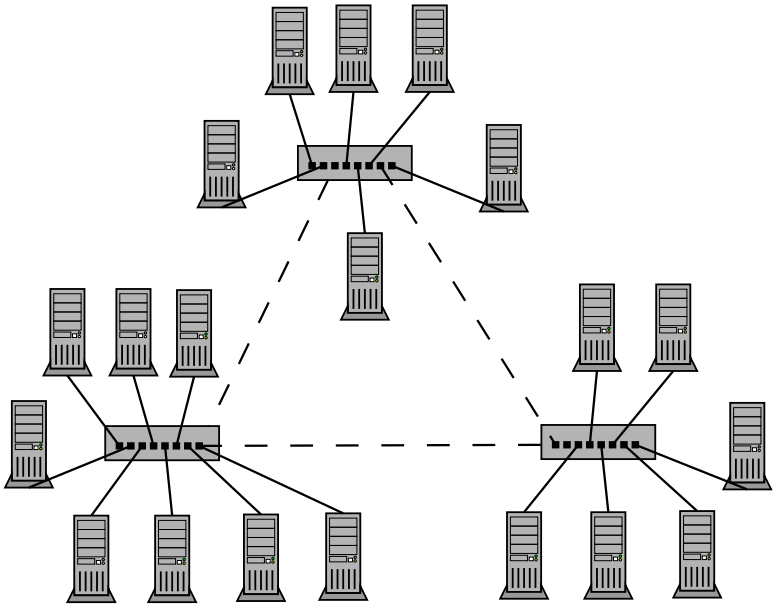

FIG. 3.2 – Une fédération de grappes de calculateurs

Hypothèses sur le réseau Nous supposons qu'en l'absence de défaillance de nœuds, le réseau est fiable, c'est à dire qu'un message envoyé sera reçu dans un temps arbitraire mais fini. Ceci peut être réalisé grâce à l'utilisation d'un protocole comme TCP/IP. Cette hypothèse est réaliste, et le service offert par le mécanisme de tolérance aux fautes au niveau applicatif est ainsi de meilleur qualité.

3.2.2 Modèle de fautes

Nous considérons le modèle "*fail silent*", c'est-à-dire qu'un nœud cesse de fonctionner "silencieusement". Nous ne considérons pas les fautes d'omission ou les fautes byzantines. Nous considérons qu'une grappe est hautement disponible. Autrement dit, la défaillance d'un nœud d'une grappe n'empêche pas les autres nœuds de la grappe de fonctionner. Ceci peut être assuré par des systèmes à image unique comme Kerrighed[42].
Nous considérons que dans chaque grappe un seul nœud peut être défaillant à un instant donné. De plus, nous supposons qu'il y a toujours, au sein d'une grappe, un nœud susceptible d'accueillir les processus à redémarrer (si le protocole est implanté au niveau système, cela peut signifier que le nœud sur lequel un processus est relancé possède une achitecture du même type que le nœud d'origine).

3.2.3 Modèle d'applications

Nous considérons une application parallèle comme un ensemble de processus distribués sur différents nœuds (dans notre cas, il s'agit des calculateurs des grappes de la fédération). Ces processus communiquent entre eux par des échanges de messages [1]. Nous ne faisons aucune hypothèse quand au déterminisme de l'exécution des processus. Autrement dit, à un instant *t*, il n'est pas possible de prévoir ce qu'il va se passer à l'instant *t+1*. Le non-déterminisme peut-être dû à des interruptions matérielles, des tirages aléatoires ou la prise en compte du temps. Ceci a pour concéquence que lors de deux exécutions, un même processus peut se comporter différemment (par exemple, le contenu d'un message peut être différent).
Afin de bien exploiter l'architecture de fédération de grappes, nous supposons que les processus communiquant beaucoup s'exécutent au sein d'une même grappe. Les communications inter-grappes restent ainsi limitées. Cette hypothèse est faite pour des raisons de performances, le protocole de tolérance aux fautes doit fonctionner même si elle n'est pas vérifiée.

3.2.4 Modèle de système

Dans la suite de ce document, nous utilisons le modèle de système à deux couches suivant. Les

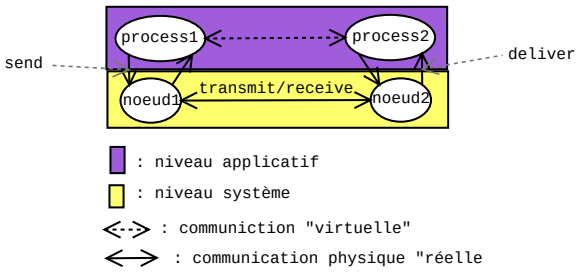

FIG. 3.3 – Modèle à deux couches

processus de l'application parallèle s'exécutent au sein de la couche applicative, le protocole de tolérance aux fautes est implanté dans la couche système. Les messages applicatifs émis par les processus sont traités par le protocole de tolérance au fautes. Cela signifie que lorsque l'application envoie un message (*send*), il est intercepté par le protocole qui peut le journaliser ou y adjoindre un numéro de séquence avant de le transmettre effectivement (*transmit*). De plus, le protocole de tolérance aux fautes peut sauvegarder l'état (mémoire, pile, fichiers ouverts...) des processus applicatifs, donc effectuer des sauvegardes d'états locaux (*takeTentativeCkPt*) ainsi que relancer ces derniers à partir de ces sauvegardes (*restart/rollback*) ou envoyer des messages protocolaires aux autres nœuds impliqués dans le calcul.

[1] Nous ne considérons pas le cas des DSM, un tel système pouvant être considéré comme une application répartie communiquant par échanges de messages, selon le niveau où l'on se place.

3.3 Principes de conception du protocole

3.3.1 Principaux problèmes posés par l'architecture et les hypothèses

Les principaux problèmes auxquels il faut faire face lors de la mise en place d'un protocole de reprise d'applications parallèles dans une fédération de grappes de calculateurs sont les suivants.
La grande échelle et les fortes latences entre les grappes rendent une synchronisation de l'ensemble des processus pour la sauvegarde d'un point de reprise coûteuse. D'autre part, étant donné que l'hypothèse que tous les évènements non-déterministes survenant au sein d'une grappe soient détectables est très forte, les mécanismes de journalisation ne doivent prendre en compte que les messages qui ne seront pas regénérés.

La grande échelle et la synchronisation

Dans le chapitre 2, nous avons vu qu'il était possible de calculer la ligne de recouvrement au moment de la création du point de reprise en utilisant la technique du point de reprise coordonné. Cependant, la coordination nécessaire à la prise de tels points de reprise nécessite un protocole de validation à deux phases au cours duquel les communications sont "gelées". En général, un nœud initiateur envoie une requête à tous les autres nœuds, qui en accusent réception, puis un message de validation leur est retourné. Dans le cas où une faute se produit durant ce protocole, des délais de garde peuvent se déclencher. Au sein d'une fédération de grappes, un tel protocole n'est guère envisageable. En effet, le nombre important de nœuds et la latence importante entre les grappes implique un temps de synchronisation long (au cours duquel les communications sont gelées) ce qui augmente la probabilité d'apparition d'une faute au cours de la sauvegarde d'un point de reprise. Cette technique de point de reprise ne passe pas à l'échelle. Dans certains cas, le temps de synchronisation pourrait dépasser le MTBF.
Nous avons vu que dans les systèmes à grande échelle, les techniques de points de reprise indépendants étaient bien adaptées.

Le non-déterminisme des applications et la journalisation

Les évènements non-déterministes sont les réceptions de messages, les intéruptions et signaux, les tirages aléatoires... Nous considérons que l'exécution d'une application n'est pas déterministe, et qu'il n'est pas envisageable de détecter et de journaliser tous les évènements non-déterministes afin de les rejouer en cas de retour arrière. Cette hypothèse est appelée hypothèse de déterminisme partiel (*piecewise deterministic assumption*). Dans la réalité, il semble que cette hypothèse soit très forte, y compris pour un seul processus (un signal peut intervenir lors d'un appel système par exemple).
Dans les fédérations de grappes, une première idée serait de journaliser les communications inter-grappes afin de pouvoir les rejouer en cas de défaillance (nous avons émis l'hypothèse que les communications de ce type étaient limitées). Cependant l'hypothèse de déterminisme partiel au sein d'une grappe est irréaliste la détection et la rejoue de tous les évènements non-déterministes au sein d'une telle architecture serait très dificile à implanter, et serait certainement inefficace (par exemple, toutes les communications intra-grappe devraient être rejouées dans le même ordre).

L'hypothèse de déterminisme n'est nécessaire que lorsque l'émetteur d'un message effectue un retour arrière en amont de l'envoi du message : dans ce cas, il va le regénérer. Sans l'hypothèse de déterminisme partiel, ce dernier peut être différent par rapport au premier envoi. Il est alors nécessaire que le récepteur effectue un retour arrière et reçoive la nouvelle version du message afin de conserver une cohérence globale. La journalisation des messages inter-grappe peut tout de même être utilisée sans l'hypothèse de déterminisme partiel : c'est ce qu'illustre la figure 2.4. Considérons un système (un nœud ou une grappe) A qui envoie un message m à un autre système (B) et que B est défaillant. Si A n'effectue pas de retour arrière, alors le message m peut être rejoué sans tenir compte d'hypothèse de déterminisme. En effet, A n'effectuant pas de retour arrière, le message m ne sera pas regénéré.

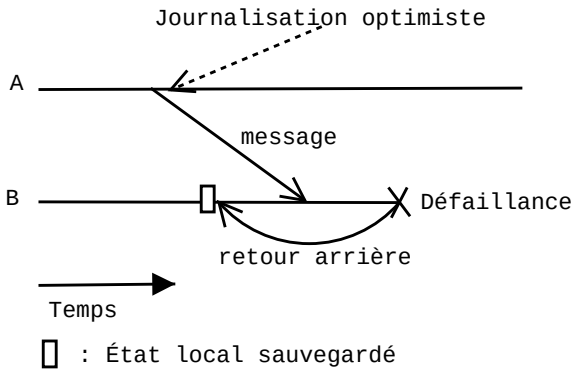

FIG. 3.4 – Journalisation sans hypothèse de déterminisme

Sur la figure la journalisation est effectuée par de l'émetteur. En effet, les journaux ne sont utilisés que si le site émetteur n'effectue pas de retour arrière. Cela permet donc de journaliser le message de manière optimiste dans un espace de stockage non stable.

3.4 Description du protocole

Cette partie décrit le protocole de reprise d'applications parallèles que nous avons conçu pour les fédérations de grappes de calculateurs. Ce protocole se veut être aussi simple que possible, le premier but est d'assurer la propriété de sûreté (i.e. assurer qu'il y ait toujours des états locaux sauvegardés permettant un retour arrière, et que lors d'un retour arrière, l'état global restauré soit cohérent). Ensuite, il doit être aussi efficace que possible. Les mécanismes de points de reprise coordonné sont intéressants de part leur simplicité. Le protocole de tolérance aux défaillances proposé se veut aussi coordonné que possible. L'idée est de relâcher cette coordinnation, uniquement entre les grappes. Le protocole combine une technique de points de reprise coordonnés au sein des grappes et induit par les communications entres les grappes. La technique de points de reprise induit implique de conservé plusieurs points de reprise dans chaque grappe, il est don nécessaire de mettre au point un ramasse-miette collectant les points de reprises devenus obsolètes.
La fédération de grappes est composée d'un certain nombre de sites (i.e grappes) qui contiennent un certain nombre de calculateurs. On considère K sites (numérotés de 1 à K), dans chaque site, il y a N_j nœuds où j est le numéro du site (numérotés de 1 à N_j).

3.4.1 sauvegarde d'un point de reprise

Point de reprise induit et journalisation des messages inter-grappes

Lorsque l'on se situe entre deux grappes, une idée intuitive est de considérer chacune de ces grappes comme un "super-nœud". Cependant, ces "super-nœuds" sont très particuliers : ils ne sont jamais complètement défaillants (nous avons fait l'hypothèse qu'une grappe était toujours disponible). En revanche, les processus s'exécutant sur une grappe peuvent effectuer des retours arrière (le MTBF d'une grappe est inférieur au minimum des MTBF des nœuds). De plus, nous avons vu que l'hypothèse de déterminisme partiel devient très forte (voir irréaliste). Le premier des deux points ci-dessus implique qu'une technique de point de reprise coordonné entre ces super-nœuds est inefficace. Le dernier implique que la journalisation des messages échangés entre deux super-nœuds nécessiterait des mécanismes permettant de rejouer une exécution parallèle au sein d'une grappe, ce qui semble très difficile à implanter dans la réalité. Ce dernier point empêche d'utiliser une technique de points de reprise indépendants entre les grappes avec des stratégies de journalisation pour limiter l'effet domino.
Une manière d'effectuer des points de reprise *relativement indépendants* sans tenir compte de l'hypothèse de déterminisme est d'utiliser une méthode de point de reprise "quasi-synchrone". Ces méthodes correspondent aux techniques de points de reprise induits par les communications (présentées au premier chapitre). L'effet domino est éliminé en forçant la sauvegarde de certains points de reprise lors des communications en fonction d'informations ajoutées aux messages.
Dans chaque grappe, des points de reprise coordonnés non-forcés sont sauvegardés périodiquement de manière indépendante des autres grappes. Lors d'une communication inter-grappe, la grappe dans laquelle se situe le nœud récepteur peut être forcée de prendre un point de reprise. Tous les points de reprise intra-grappe sont des points de reprise coordonnés.
Les techniques de points de reprise induits par les communications ont été classifiées dans [27]. Cette classification tient essentiellement compte du nombre de points de reprise forcés "inutiles". Forcer un point de reprise avant chaque réception de message empêche bien l'effet domino, mais ajoute un surcoût important et inutile. Dans la figure 3.5 illustrant une exécution répartie sur trois processus, le point de reprise 1,2 est inutile. En effet, même si *p1* est défaillant après avoir émis *m2*, il retournera en arrière jusqu'au point de reprise 1,1 de manière à conserver la cohérence en tenant compte de *m1*. Nous nous intéressons donc aux conditions permettant de prendre un nombre minimum de

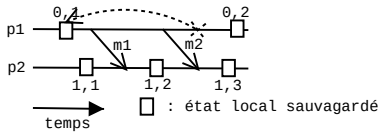

FIG. 3.5 – Point de reprise forcé inutile

points de reprise forcés. [31] définit les "Z-chemins" qui sont une extension de la relation causale [7]. Dans la figure 3.6, il y a un Z-chemin entre les états e1 et e2. En effet, *m2* crée une dépendance causale entre *p3* et *p2*, et *m1* entre *p2* et *p1*. Détecter les Z-chemins est plus efficace que de détecter les dépendances causales. Dans [27] il y a une idée de la preuve que l'enregistrement les dépendances causales directes entre les nœuds peut suffir à détecter les Z-chemins.
Nous allons donc utiliser un vecteur de dépendance directe (*Direct Dependance vector* ou DDV) comme décrit dans [2], afin de détecter les Z-chemins entre les grappes et de forcer la sauvegarde

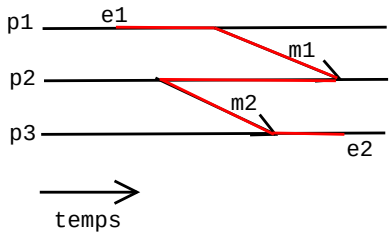

FIG. 3.6 – Z-chemin

de points de reprise de manière appropriée. La gestion des *DDV* implique que chaque grappe gère un numéro de séquence pour numéroter ses points de reprise. Le DDV d'une grappe est un vecteur dont chaque entrée est le dernier numéro de séquence de la grappe voisine correspondante (chaque grappe incrémentant un numéro de séquence à chaque fois qu'un point de reprise est établi). Ce numéro de séquence (noté *sn*) est ajouté à chaque message inter-grappe.
Lors de la réception d'un message provenant d'un autre site (dans ce document les termes "site" et "grappe" sont équivalents), le numéro de séquence qu'il contient est comparé avec l'entrée du DDV correspondante. Si il est supérieur, un point de reprise est forcé avant de tenir compte du message. Quand un nœud dans une grappe est défaillant, la grappe effectue un retour arrière et alerte les autres grappes. Lorsqu'une grappe reçoit une telle alerte (contenant le numéro de séquence du point de reprise restauré sur le site émetteur) le DDV est utilisé afin de voir si le deuxième site doit lui aussi effectuer un retour arrière. Le deuxième site effectue un retour arrière si et seulement si il possède un point de reprise pour lequel l'entrée de son DDV correspondant au site ayant envoyé l'alerte est supérieure ou égale au numéro de séquence reçu.
De plus il est possible de limiter le nombre de sites devant effectuer un retour arrière en utilisant un mécanisme de journalisation. [19] montre qu'il y a un compromis à faire entre le nombre de points de reprise forcés et le nombre de messages journalisés. Nous avons vu qu'il est possible de journaliser et de rejouer les messages sans hypothèse de déterminisme si l'émetteur n'effectue pas de retour arrière en amont de l'émission. Cela signifie que les messages journalisés ne sont utilisés que si leur récepteur effectue un retour arrière et non leur émetteur. En effet, si l'émetteur effectue un retour arrière, il reproduit les messages, et rien ne garantit qu'ils soient identiques aux premiers. Dans un tel cas, les journaux ne doivent pas être utilisés. En tenant compte de cette remarque, il semble judicieux d'effectuer la journalisation côté émetteur, en effet, elle peut être faite de manière optimiste et dans un espace de stockage volatile. Chaque message envoyé vers une autre grappe est journalisé et associé au numéro de séquence du destinataire lors de la réception du message. Un acquittement permet de connaître ce numéro de séquence (tant que l'acquittement n'est pas reçu, le numéro associé correspond à l'infini). Ceci permet de déterminer quels sont les messages à rejouer (ceux dont le numéro de séquence associé est supérieur au point de reprise auquel le site défaillant effectue un retour arrière).

Point de reprise coordonné intra-grappe

Au sein d'une grappe de calculateurs, un mécanisme de point de reprise coordonné semble raisonnable. En effet, le surcoût induit par une synchronisation est relativement faible grâce aux performances des réseaux de type SAN (faible latence et débit élevé). Périodiquement, un des nœuds de

la grappe va devient l'initiateur d'un point de reprise coordonné intra-grappe. Il envoie à tous les nœuds de la grappe une requête d'établissement de point de reprise (*broadCastReqCkpt()*). Lorsqu'un nœud reçoit une requête d'établissement de point de reprise, il sauvegarde son état local *takeTentativeCkPt()*. Une fois que l'état local est sauvegardé sur espace de stockage stable (c.f. partie suivante) il envoie un acquittement à l'émetteur (*sendCkPtAck*). Lorsque l'initiateur a reçu un acquittement de la part de tous les autres nœuds de sa grappe, il valide son propre état local et envoie un message de validation à tous les autres nœuds afin qu'ils fassent de même (*broadCastCkPtCommit()*).
Afin de prendre en compte les défaillances qui peuvent survenir durant ce protocole de validation à deux phases, des délais de garde sont déclenchés, il y a donc des bornes de temps entre l'émission de la requête et la réception des accusés, ainsi qu'entre l'émission d'un accusé de réception et la réception d'une validation. Si ces bornes de temps sont dépassées, l'état sauvegardé est annulé.
Il est possible que deux nœuds décident "en même temps" de déclencher la sauvegarde d'un point de reprise coordonné, dans ce cas, le rang du nœud est utilisé pour déterminer lequel est considéré comme l'initiateur. Les autres nœuds peuvent donc recevoir plusieurs requêtes et renvoyer un ou plusieurs acquittements en fonction du moment de la réception des requêtes et des rangs des nœuds émetteurs de ces dernières (c.f. algorithme 4).

Espace de stockage stable En cas de défaillance d'un nœud, les processus qui s'exécutent sont relancés à partir d'un point de reprise sauvagardé, ces derniers doivent donc être conservés dans un espace de stockage stable (c.f. chapitre 1). Étant donné que l'on considère qu'une seule défaillance peut intervenir simultanément au sein d'une même grappe, cet espace de stockage peut être implanté en utilisant les mémoires de deux nœuds distincts de la grappe. Ceci permet de limiter les accès aux disques qui sont moins performants que les accès réseaux et mémoires au sein d'une grappe. De manière à limiter le traffic réseau, les états locaux sont sauvegardés en mémoire sur les nœuds eux-mêmes, ainsi que sur leur voisin [2]. Ainsi un seul état local par nœud est envoyé sur le réseau.
En implantant l'espace de stockage stable de cette manière, un nœud non défaillant devant effectuer un retour arrière a toujours accès aux points de reprise conservés localement. De même un nœud défaillant lors de son redémarrage peut demander les poitns de reprise des processus qu'il exécutait à son voisin.
Lorsqu'un nœud reçoit une requête de prise de point de reprise coordonné, il sauvegarde son état localement et en envoie une copie à son nœud voisin. Quand il reçoit un acquittement lui signalant que son état local a bien été sauvegardé à distance, il envoie un acquittement à l'émetteur de la requête de sauvegarde de point de reprise.

3.4.2 Rôles

Certains nœuds jouent un rôle particulier : dans chaque grappe, nous choisissons certains nœuds que l'on appellera *leaders*, ce sont eux qui auront la charge de détecter les défaillances, de redemmarrer les autres nœuds et des communications inter-grappes concernant la ligne de récupération et le ramasse miette.

[2] Les nœuds étant numérotés, le voisin est le nœud ayant le numéro suivant modulo le nombre de nœuds de la grappe.

De temps en temps un nœud devient un *initiateur de point de reprise*, cela signifit qu'il va diffuser une requête dans son site et se préocuper de sa validation.

3.4.3 Variables et structures de données

Constantes
- $nbSites$ nombre de sites (i.e. K).
- $mySiteId_i$ identifiant de site.
- $nbNodes_i$ nombre de nœuds dans le site i.
- $myRank_{i,j}$ identifiant du noeud j dans le site i.
- $lSet_i$ ensemble des leaders du site i.
- $otherLeaders_i$ ensemble des leaders des autres sites - dans chaque site, les leaders doivent pouvoir comuniquer avec les leaders des autres sites.

Délais de garde
- $iMALIVETimer$ pour envoyer des messages périodiques du type *"je suis en vie"*.
- $chCkAliveTimer$ pour vérifier si les sites sont tous présents.
- $tentativeCkPtTimer$ temps maximum entre la sauvegarde d'un point de reprise et sa validation.
- $waitForAllTimer$ temps maximum entre la diffusion d'une requête de prise de point de reprise et la réception de tous les accusés.
- $gCTimer$ pour déclencher le ramasse miettes.
- $ckPtTimer$ temps entre deux points de reprise non forcés.

Autres
- $mySn_{i,j}$ le numéro de séquence incrémenté à chaque validation de point de reprise (au sein d'une même grappe, en dehors du protocole de validation à deux phases, il est identique sur chaque nœud).
- $myDDV_{i,j}$ Le vecteur des dépendances directes (synchronisé sur chaque noeud lors de l'établissement de point de reprise).
- $duringCkPt_{i,j}$ un booléen permettant de savoir si un nœud se situe au sein du prococole de validation du point de reprise coordonné ou non.
- $hb_{i,j}$ un vecteur de taille $nbNodes_i$ dont chaque entrée est le compte des messages "je suis en vie" reçus par le nœud correspondant.
- $oldHb_{i,j}$ une copie du vecteur précédent, permettant de visualiser l'évolution du nombre de messages reçus.
- $ckPtAckSet_{i,j}$ permet à un initiateur de point de reprise de stocker l'ensemble des nœuds ayant accusé de réception la requête.
- $gcAckSet_{i,j}$ permet à un initiateur de point de collection (ramasse miettes) de stocker l'ensemble des nœuds ayant accusé de réception la requête.
- $initiator_{i,j}$ rang du dernier initiateur de point de reprise.

Journaux Chaque nœud conserve dans un journal en mémoire vive les messages émis par les processus applicatifs, leur destinataire et leur numéro de séquence (connu grâce à un acquittement).

3.4.4 Initialisation

L'algorithme suivant est exécuté sur chaque nœud de la féderation afin d'initialiser les structures de données du protocole.

Algorithme 1: initialisation

```
for i ← 0 to nbSites do
    myDDV[i] ← 0;
mySn ← 0;
duringCkPt ← false;
launchTimer(iMALIVETimer );
launchTimer(ckPtTimer );
if ROLE = leader then
    launchTimer(chCkAliveTimer );
```

3.4.5 Messages échangés

Les messages échangés au cours de l'exécution du protocole ont la structure suivante:
- *sender* l'émetteur du message, c'est à dire
  - *sender.rank* son rang
  - *sender.siteId* l'identifiant de son site (sa grappe).
- *type* le type du message (c.f. l'algorithme : *Message dispatching*).
- *subtype* le sous-type du message (c.f. l'algorithme : *ckPtHandler*).
- *sn* le numéro de séquence de l'émetteur.
- *data* le message.

L'algorithme 2 présente l'aiguillage des messages en fonction de leur type et de leur émetteur à leur réception sur un nœud.

Algorithme 2: Aiguillage des messages

```
Data      : m, the received message
if m.sender.siteId=mySiteId then
    message from the cluster;
    switch m.type do
        case CKPT
            ckPtHandler(m );
        case ROLLBACK
            rollbackHandler(m );
        case FD
            fdHandler(m );
        case GC
            gcHandler(m );
        case INTERNALALERT
            internalAlertHandler(m );
        otherwise
            just a normal application message from the same cluster -> nothing to do;
            if duringCkPt =true then
                storeMessage(m );
            else
                deliver(m );
else
    message from another cluster in the federation;
    if duringCkPt =true then
        let the coordinated checkpoint finish;
        storeMessFromOut(m );
    else
        switch m.type do
            case ROLLBACKALERT
                the node is part of its clusters lSet;
                rollbackAlertHandler(m );
            case ACK
                ackFromOutsideHandler(m );
            otherwise
                just a normal application message;
                messFromOutsideHandler(m );
```

3.4.6 Algorithmes de sauvegarde de points de reprise

Algorithme 3: initiateCkPt

```
initialization of some data structures;
ckPtAckSet ← ∅;
duringCkPt ← true;
initiator ← myRank;
take the node's own tentative checkpoint;
takeTentativeCkPt();
ask the other nodes in the cluster to do the same;
broadCastReqCkPt();
in case of failure during the checkpoint;
launchTimer(waitForAllTimer );
```

Déclenchement d'un point de reprise coordonné.

Algorithme 4: ckPtHandler

```
Data      : m received from a node in the cluster
switch m.subType do
    case REQ
        the node is requested to take a tentative checkpoint;
        if duringCkPt =false then
            Not currently checkpointing;
            stopTimer(ckPtTimer ) don't initiate a new checkpoint;
            initiator ← m.sender.rankremember the initiator's rank;
            duringCkPt ← true;
            takeTentativeCkPt();
            sendCkPtAck(m.sender,myDDV ) acknowledgement;
            launchTimer(tentativeCkPtTimer );
        else
            the node is already in a checkpoint phase;
            if m.sender.rank<initiator then
                only the one with the smallest rank is taken into account;
                if initiator =myRank then
                    the node was the other initiator;
                    stopTimer(waitForAllTimer );
                    removeCkPtAckSet();
                else
                    the node was not the other initiator;
                    stopTimer(tentativeCkPtTimer );
                initiator ← m.sender.rank;
                sendCkPtAck(m.sender,myDDV );
                launchTimer(tentativeCkPtTimer );
    case ACK
        If we receive an Ack, we are the initiator;
        add(ckPtAckSet,m.sender,receivedDDV );
        if ckPtAckSet =ALLINGRP then
            stopTimer(waitForAllTimer );
            computeNewDDV generate a new DDV in which entries are the max of the
            corresponding entries in all the DDVs;
            computeNewDDV(ckPtAckSet,newDDV );
            makeTentativePermanent();
            duringCkPt ← false;
            broadCastCkPtCommit(newDDV );
    case COMMIT
        stopTimer(tentativeCkPtTimer );
        makeTentativePermanent() this also increment mySn;
        myDDV ← newDDV;
        duringCkPt ← false;
        deliverAll() deliver all the waiting messages;
        replayMessFromOut();
        sendAll() send all the waiting messages;
        launchTimer(ckPtTimer );
```

Traitement des messages de point de reprise. Notons que la synchronisation induite par le protocole de validation à deux phases assure également que le DDV soit identique sur chaque nœud du site (c.f. explications).

3.4.7 Algorithmes d'émission et réception des messages applicatifs

Algorithme 5: send

```
Data      : mess, the sent message
the message type and subtype are set by the function that call send, for example sendCkPtAck
will set type to CKPT and subtype to ACK;
if duringCkPt =true then
    froze communications during checkpointing;
    storeToSend(mess );
else
    mess.sn ← mySn;
    mess.sender.rank ← myRank;
    mess.sender.siteId ← mySiteId;
    send the message on the network;
    transmit(mess );
    if it's an inter-cluster communication, log it;
    if receiver.siteId ≠ mySiteId then
        the logging is optimistic and doesn't need stable storage;
        logVolatile(mess );
        its sn is suppose to be infinite (i.e. will have to be replayed) until the acknowledgment;
        logVolatile(∞);
```

Algorithme 6: messFromOutsideHandler

```
It comes from another site, check the dependences;
if m.sn>myDDV [m.sender.siteId] then
    acknowledge the message with the next sequence number: a checkpoint will be taken;
    acknowledgeMess(mySn ++);
    update the DDV;
    myDDV[m.sender.siteId] ← m.sn;
    the message will be delivered at the commit;
    storeMessage(m );
    initiateCkPt();
else
    acknowledge the message with the current sequence number;
    acknowledgeMess(mySn );
    deliver(m );
```

Algorithme 7: ackFromOutsideHandler

an inter-cluster send is acknowledged with the receiver current sn in the message data;
`logVolatile(`*m.sn* `)`;
the received sn will replace the ∞ *stored during the message logging phase*;

3.4.8 Retour arrière

Quand une faute est détectée, les nœuds de la grappe doivent restaurer un point de reprise de l'application sauvegardé et validé.[3]. Le nœud détectant la faute envoie une requête de retour arrière à tous les nœuds de la grappe et redémarre le processus fautif. De plus, les autres grappes de la fédération sont prévenues de ce retour arrière (elles doivent en effet éventuellement revenir en arrière et/ou rejouer des messages). Lorsque qu'un nœud est défaillant, il se produit un retour arrière de l'application au sein de la grappe. Afin de conserver la cohérence inter-grappes, les autres doivent être prévenues de ce retour arrière.

Algorithme 8: rollbackAlertHandler

sn has been received from m.sender.siteId;
if $\exists$ *ckPt so that myDDV*$[m.sender.siteId] \geq sn$ **then**
  FirstCkPt returns the sn of the first checkpoint that has "m.sn" equal or greater in its "mess.sender.siteId" DDV entry;
  newSn $\leftarrow$ `firstCkPtSn(`*m.sender.siteId,sn*`)`;
  `broadcastRollback(`*newSn* `)`;
  `alertRollback(`*otherLeaders,newSn* `)`;
tell every one in the cluster to watch if there is something to re-send;
`internalAlert(`*m.sender.siteId,sn* `)`;

Traitement des messages d'allerte de retour arrière. Après la défaillance d'un nœud dans la fédération, il est possible qu'un site reçoive plusieurs alertes, dans ce cas, il effectuera le retour le plus pessimiste.

Algorithme 9: internalAlertHandler

if there're logs for a node in the site with log.sn $\geq$ *received sn re-play them*;
if $\exists$ *logs so thatlog.siteId* $=$ *receivedSiteId and log.sn* $\geq$ *receivedSn* **then**
  `replay(`*receivedSiteId,receivedSn* `)`;

Diffusion des alertes de retour arrière au sein des sites afin que les messages soient rejoués.

[3]Dans les fédérations de grappes de calculateurs, une grappe devra éventuellement revenir à des points de reprises en amont de manière à prévenir les dépendances inter-grappes.

3.4.9 Ramasse miettes

Étant donné que l'on utilise une technique de points de reprise induits par les communications, il est possible que les nœuds d'un site doivent revenir en arrière en amont du dernier point de reprise sauvegardés. Il est donc nécessaire de conserver sur les nœuds tous les points de reprise qui sont suceptibles d'être restaurés. De même, les messages journalisés suceptibles d'être ré-émis doivent être stockés.
Il est nécessaire de supprimer les données devenues inutiles. Afin de limiter l'espace de stockage utilisé par le protocole.
Le ramasse-miettes proposé est centralisé. Un nœud dans une grappe déclenche la collecte des données obsolètes. Cela peut être fait périodiquement ou lorsque le nœud est surchargé. Le nœud initiateur de la collecte demande à un nœud de chaque site la liste des DDV associés aux points de reprise. Toutes les listes des DDV sont utilisées pour calculer le vecteur de retour arrière "au pire". Celui-ci est un vecteur dont chaque entrée est le numéro de séquence auquel chaque site peut retourner "au pire". Il est envoyé à chaque site où il est difusé à tous les nœuds. Ces derniers peuvent alors épurer leur journal et les points de reprises devenus obsolètes.
Sur un site i, les points de reprise obsolètes sont ceux dont le numéro de séquence associé est inférieur à l'entrée du vecteur correspondant au site i. Pour chaque grappe voisine, les messages à supprimer des journaux sont ceux dont le récepteur appartient à cette grappe et dont le numéro de séquence associé est inférieur à l'entrée du vecteur correpondant a ce site voisin.

Algorithme 10: gcHandler

```
switch m.subtype do
    case REQ
        sendGCACK(myDDV );
    case ACK
        add(gcAckSet,m.sender,receivedDDV );
        if gcAckSet.size=nbSites then
            the initiator has received all the DDVs;
            stopTimer(gCTimer );
            computeRecoveryLine(gcAckSet,recoveryLine );
            sendCollect(otherLeaders,recoveryLine );
    case COLLECT
        if ROLE = _LEADER then
            broadcastCollect(recoveryLine );
        clean(recoveryLine );
```

Algorithme de ramasse miette.

Détection d'erreurs

L'étude des détecteurs de défaillance dépasse le cadre de ce rapport. Cependant, un système simple d'envoi de messages de type "je suis en vie" périodiques et de comptage de ces derniers a été mis en place dans les algorithmes du protocole, ainsi que dans le simulateur à évènements discrets.
Chaque nœud envoie périodiquement des messages "je suis en vie" à deux *leaders* au sein de leur grappe, ces messages sont comptabilisés. De temps en temps, les *leaders* vérifient qu'au moins un message "je suis en vie" a été reçu de chaque nœud au cours d'un certain laps de temps. Dans le cas contraire, le nœud est considéré comme fautif.

Algorithme 11: fdHandler

```
increment the heartbeat of the node that sends a "I am alive" message;
hb[m.sender.rank] ← hb[m.sender.rank] + 1;
```

Algorithme 12: timerHandler

```
switch timer do
    case tentativeCkPtTimer
        duringCkPt ← false;
        removeTentativeCkPt();
    case waitForAllTimer
        duringCkPt ← false;
        removeTentativeCkPt();
        initiateCkPt();
    case chCkAliveTimer
        launched only on the leaders;
        oldHb ← hb;
        for i ← 0 to nbNodes do
            if oldHb =hb then
                node i has failed;
                broadCastRollBack(mySn );
                restart(i,mySn );
                alertRollback(otherLeaders,mySn );
    case iMALIVETimer
        sendFd(lSet );
    case ckPtTimer
        initiateCkPt();
    case gCTimer
        re-initialize data structures, re-launch the timer and send requests;
        restartGarbageCollection();
```

Réception d'un message de type "je suis en vie".

3.4.10 Propriétés

Le début de l'exécution d'un processus (i.e. son état initial) est considéré comme le premier point de reprise. Le numéro de séquence associé à ce premier point de reprise est 0. Sur chaque nœud, la valeur de $mySn$ est égale à la valeur du dernier point de reprise validé. Entre le point de reprise x et le point de reprise y la valeur de $mySn$ est x. On dit que le nœud est à l'époque x. Quand un état local est sauvegardé, la valeur associée est le numéro de séquence courant plus un. Le numéro de séquence est incrémenté au moment de la validation.

Le numéro de séquence : mySn Au sein d'un site, tous les nœuds ont la même valeur pour $mySn$ en dehors du protocole de validation à deux phases. Au cours du protocole de validation, deux nœuds peuvent avoir deux valeurs différentes (cette différence ne dépasse pas une unité).
Au début cette propriété est vraie : tous les $mySn$ sont égaux à 0. Ensuite, cette variable n'est modifiée (par incrémentation) que si un point de reprise est validé. Si un point de reprise a été validé au sein d'un site et s'il n'y a pas de défaillance, alors tous les nœuds reçoivent une validation et incrémentent leur variable $mySn$. S'il y a une défaillance, alors tous les nœuds effectuent un retour arrière jusqu'au point de reprise portant le numéro de séquence qui leur est ordonné par diffusion (et qui est donc identique pour tous) et qui devient leur $mySn$.

Le vecteur de dépendances directes : myDDV À chaque fois que le DDV est modifié sur un nœud (lors de la réception d'un message provenant d'un autre site), ceci implique la sauvegarde forcée d'un point de reprise. Le DDV a la même valeur sur chaque nœud au sein d'une grappe en dehors du protocole de validation à 2 phases (il est synchronisé durant la coordination). Le numéro de séquence d'un nœud et l'entrée du DDV correspondant au site du nœud sont redondants.
Comme le numéro de séquence et le DDV ne sont pas représentatifs au cours du protocole de validation à deux phases, toutes les communications inter et intra grappes sont gelées au cours de cette période.

Journalisation des messages Les messages sont journalisés côté émetteur de manière optimiste (ils ne sont rejoués que si l'émetteur n'effectue pas de retour en amont de l'émission). Ils sont journalisés avec le numéro de séquence qu'a le récepteur lors de la réception du message. Au moment de l'envoi, il sont associés avec ∞ comme numéro de séquence jusqu'à ce qu'un acquittement du récepteur vienne renseigner la valeur réelle. Si le récepteur effectue un retour arrière plus en amont que l'émetteur, ce dernier ré-émettra les messages dont le numéro de séquence est supérieur ou égal à celui auquel le récepteur est retourné (ceux qui ont été envoyés après le point où l'émetteur effectue un éventuel retour arrière sont éliminés lors du retour arrière et seront regénérés par la nouvelle exécution).

Cohérence Premièrement, définissons la cohérence entre deux états locaux sauvegardés par le fait qu'il n'y ait pas de messages fantômes (c.f. chapitre 1), et que les messages en transit soient journalisés et rejouables.

Au sein d'un site, les états locaux sauvegardés qui sont associés au même numéro de séquence sont cohérents. Ils ont été créés au cours d'une synchronisation durant laquelle les communications étaient gelées (assurant qu'il n'y a ni message en transit, ni message fantôme). Quand un site effectue un retour arrière, tous les nœuds restaurent des points de reprise portant un même numéro de séquence. L'état de l'application au sein du site reste donc cohérente.
Il y a toujours un état global cohérent sauvegardé (ne serait ce que l'état initial). Prendre des point de reprise forcés permet de faire progresser la ligne de récupération. Lors d'un retour arrière, il faut trouver la ligne de récupération. Quand un site x effectue un retour arrière, il envoie son nouveau numéro de séquence snx. Dans le cas où x a envoyé des messages après ou durant l'époque snx ces messages deviennent des messages fantômes. Mais avec ces messages, il avait envoyé son $mySn$ assurant que les récepteurs auraient au moins un point de reprise auquel ils auront a revenir quand ils recevront l'alerte avec snx. Ceci évite le messages fantômes. D'un autre coté, le site x peut avoir reçu des messages durant ou après l'époque snx qui deviennent des messages en transit. Si l'émetteur revient en arrière, ils seront regénérés, sinon, ils sont journalisés avec un numéro de séquence supérieur ou égal à snx et seront donc rejoués. Ceci interdit les messages en transit.

3.4.11 Exemples de scénarios

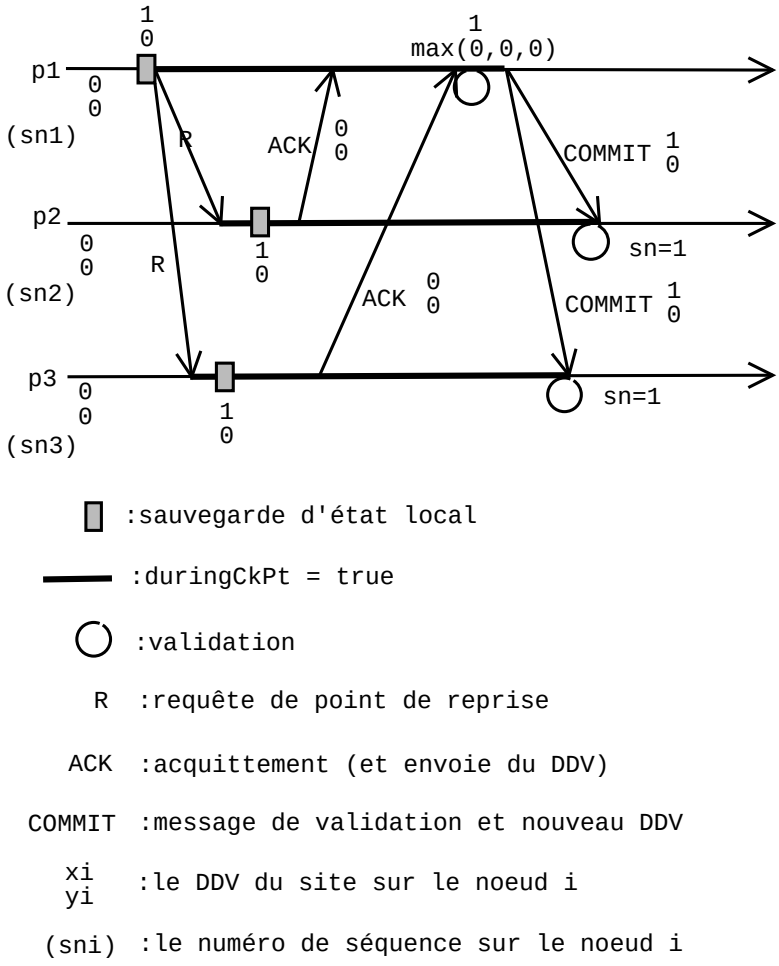

FIG. 3.7 – Point de reprise coordonné

Point de reprise intra-grappe La figure 3.7 montre le déroulement de la création d'un point de reprise en l'absence de défaillance durant le protocole de validation et en présence d'un unique initiateur au sein d'une grappe de trois nœuds. Ce site possède le numéro 0, donc toutes les variables x_1, x_2, x_3, sn_1, sn_2 et sn_3 ont la même valeur. Tous les 'sn' sont incrémentés à la fin du protocole de validation à 2 phases. Ils ont donc tous la même valeur. Le nouveau DDV est calculé par p_1, puis envoyé à tous les nœuds, qui ont donc tous le même DDV.

Deux initiateurs concurrents Quand un nœud reçoit une requête de prise de point reprise, il sauvegarde le rang de l'initiateur. Si un nœud reçoit plus d'une requête, il envoie un acquittement à celui qui a le plus petit rang. L'initiateur est celui qui a le plus petit rang. Le *faux* initiateur supprime son *ckptAckSet*, et réinitialise ses délais de garde. Le point de reprise coordonné est validé seulement par le véritable initiateur (celui qui a le plus petit rang).

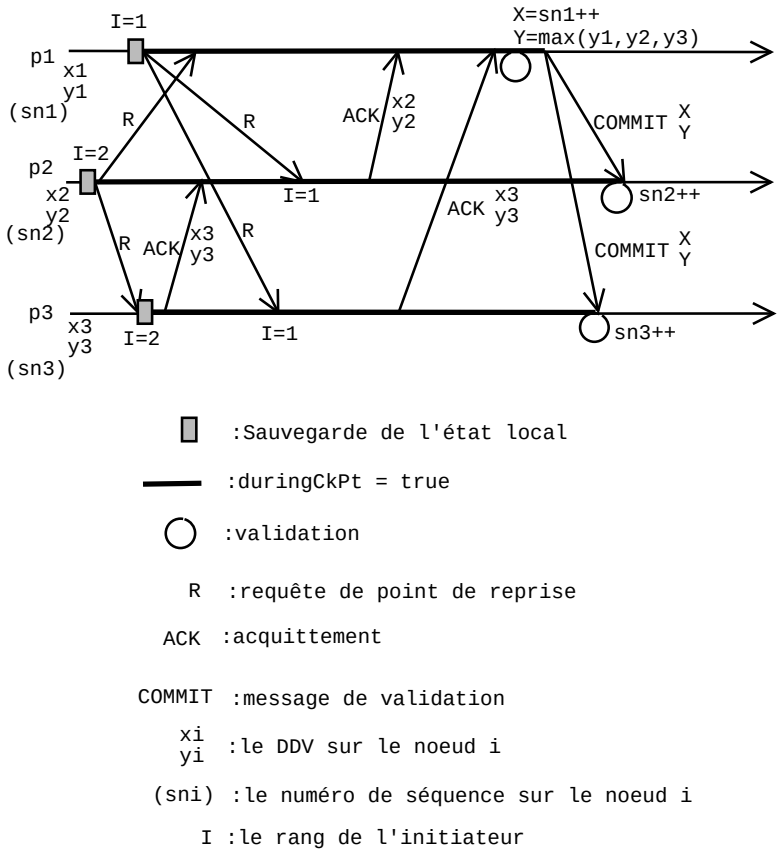

FIG. 3.8 – Deux initiateurs concurrents

Sur la figure 3.8, les nœuds 1 et 2 initialisent un point de reprise. Lorsque le nœud 2 reçoit la requête du nœud 1, il l'acquitte. Le nœud 3 reçoit d'abord la requête du nœud 2, il l'acquitte puis celle du nœud 1. Le nœud 1 est alors considéré comme le nouvel initiateur par le nœud 3 qui lui envoit donc un acquittement.

Défaillance au cours du protocole de validation à deux phases Le nœud qui détecte la défaillance demande aux autres nœuds de faire un retour arrière à son numéro de séquence courant. Si ce nœud a validé le point de reprise courant, il va demander un retour arrière à ce niveau (son *sn* a été incrémenté au moment de la validation). Le processus defaillant redémarre à partir de son dernier point de reprise non validé, mais qui avait était sauvegardé (car il a été validé sur certains nœuds).
Si le point de reprise courant n'a pas été validé sur le nœud détecteur, le numéro de séquence sur ce nœud n'a pas été incrémenté cette fois, il demande donc un retour au niveau du point de reprise précédent (même si un état à été sauvegardé sur le nœud défaillant).

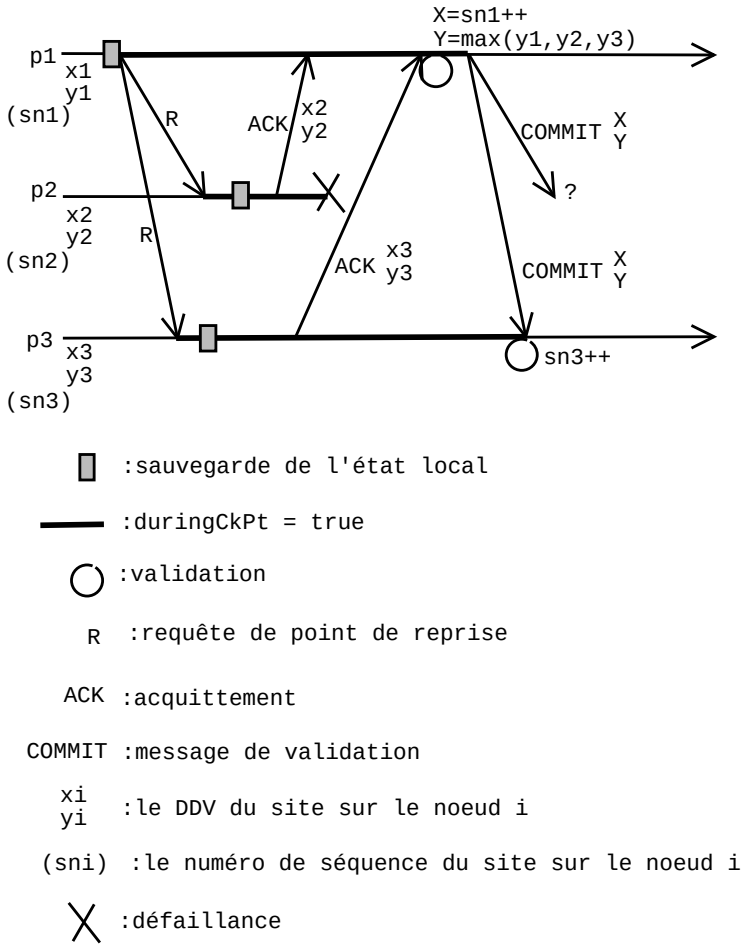

FIG. 3.9 – Défaillance au cours du protocole de validation à deux phases

Points de reprise forcés Le numéro de séquence des sites est envoyé avec les messages, il est comparé avec l'entrée correspondante du DDV du côté du récepteur, qui décide ou non de prendre un point de reprise (un point de reprise est pris si le numéro de séquence reçu est supérieur à l'entrée correspondante dans le DDV). Les acquittements ne sont pas représentés.

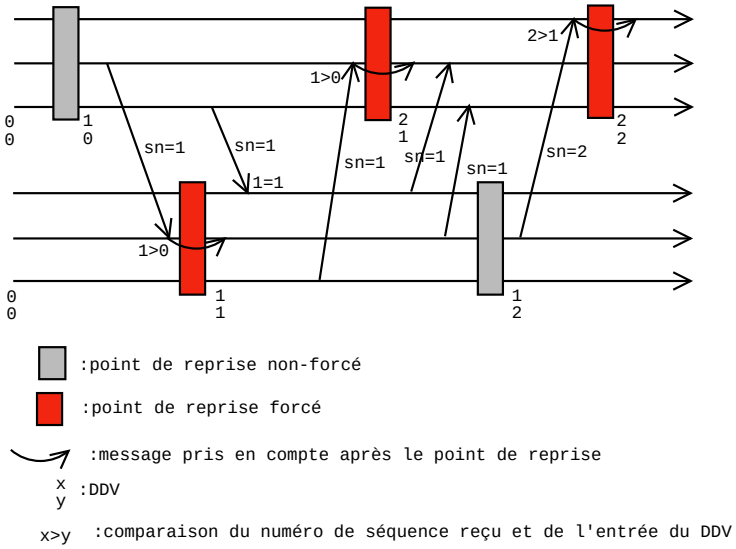

FIG. 3.10 – Points de reprise forcés

Point de reprise induit par les communications et défaillance Les nœuds du site 2 n'ont pas envoyé de messages aux nœuds du site 1, un nœud du site 2 est défaillant, impliquant un retour arrière de tous le nœuds du site 2. Le site 1 n'a pas besoin d'effectuer un retour arrière, mais il a à réémettre certains messages (il n'a pas de points de reprise avec $DDV[2] \geq 1$ mais il a journalisé des messages dont le récepteur est le site 2 et dont le numéro de séquence associé est$\geq$1). Le message 1 a été acquitté avec le numéro de séquence 1 (il a été reçu à l'époque 0 mais est pris en compte à l'époque 1 donc acquitté avec $mySn++$); le message 2 n'a pas été acquitté (donc il est journalisé avec sn=∞). En recevant l'alerte de retour arrière du site 2, les nœuds du site 1 parcourent leur journal et rejouent tous les messages qui ont un numéro de séquence associé supérieur ou égal à 1 (numéro de séquence auquel le site 2 a effectué un retour arrière).

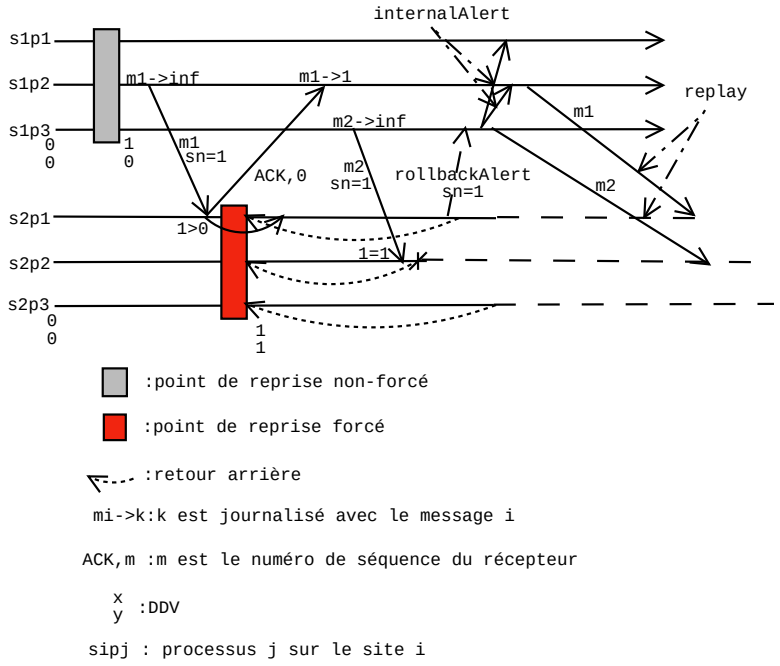

FIG. 3.11 – Point de reprise induit par les communications et défaillance

Retours arrières multiples Il y a 3 sites (les nœuds des sites ne sont pas représentés) qui ont chacun sauvegardé un point de reprise (et incrémenté leur variable *mySn*) au début de l'application. Les trois sites communiquent et prennent des points de reprise comme décrit sur la figure 3.12. Un nœud du site 3 est défaillant, impliquant un retour arrière du site 3. Une alerte de retour arrière est envoyée avec le numéro de séquence 4, le site 2 n'a pas de point de reprise où *DDV[3]* (l'entrée correspondant au site 3) est égale ou supérieure à 4, il n'a en effet pas besoin de revenir en arrière (il n'a pas reçu de message du site 3). En revanche le site 1 a un point de reprise avec $DDV = (3,0,4)$ auquel il doit revenir (sinon, en effet, m_5 deviendrait un message fatôme). Le site 1 envoie une alerte de retour arrière avec le numéro de séquence 3 mais aucun site n'aura à effectuer un retour arrière (car aucun site n'a un point de reprise avec ce numéro de séquence, 3 dans l'entrée du site ayant provoqué l'alerte, 1). De plus, le site 1 a une message (m_4) qui est journalisé avec 4 comme numéro de séquence associé, il doit le rejouer.

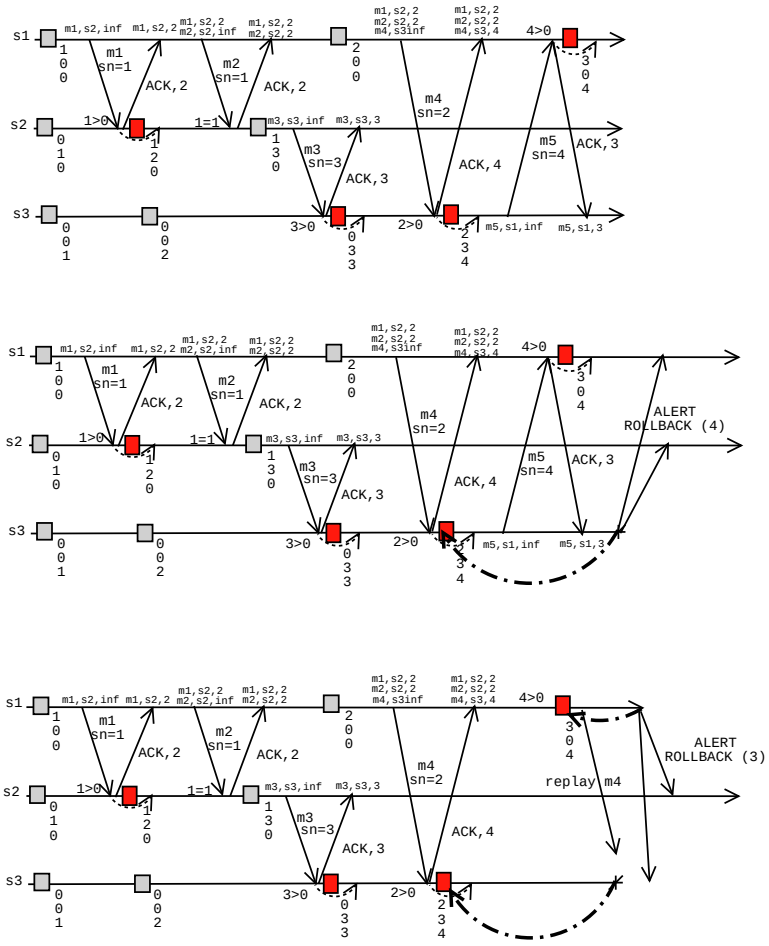

FIG. 3.12 – Retours arrières multiples

Retours arrières multiples, un autre exemple Ce dernier exemple reprend l'exécution de l'exemple précédent. Le site 2 effectue un retour arrière impliquant un retour arrière de chacun des autres sites. En effet il envoie une alerte avec le numéro de séquence 3, or, le site 3 possède un point de reprise dont le DDV associé est (0,3,3) qu'il restaure. Le site 3 envoie donc une alerte avec le numéro de séquence 3. Le site 1 possède un point de reprise dont le DDV associè est (3,0,4), qu'il doit restaurer. De plus, le site 1 devra rejouer le message *m4*. En effet, dans ses journaux (représentés sous la forme : message, site, numéro de séquence) il possède *m4,s3,4*.

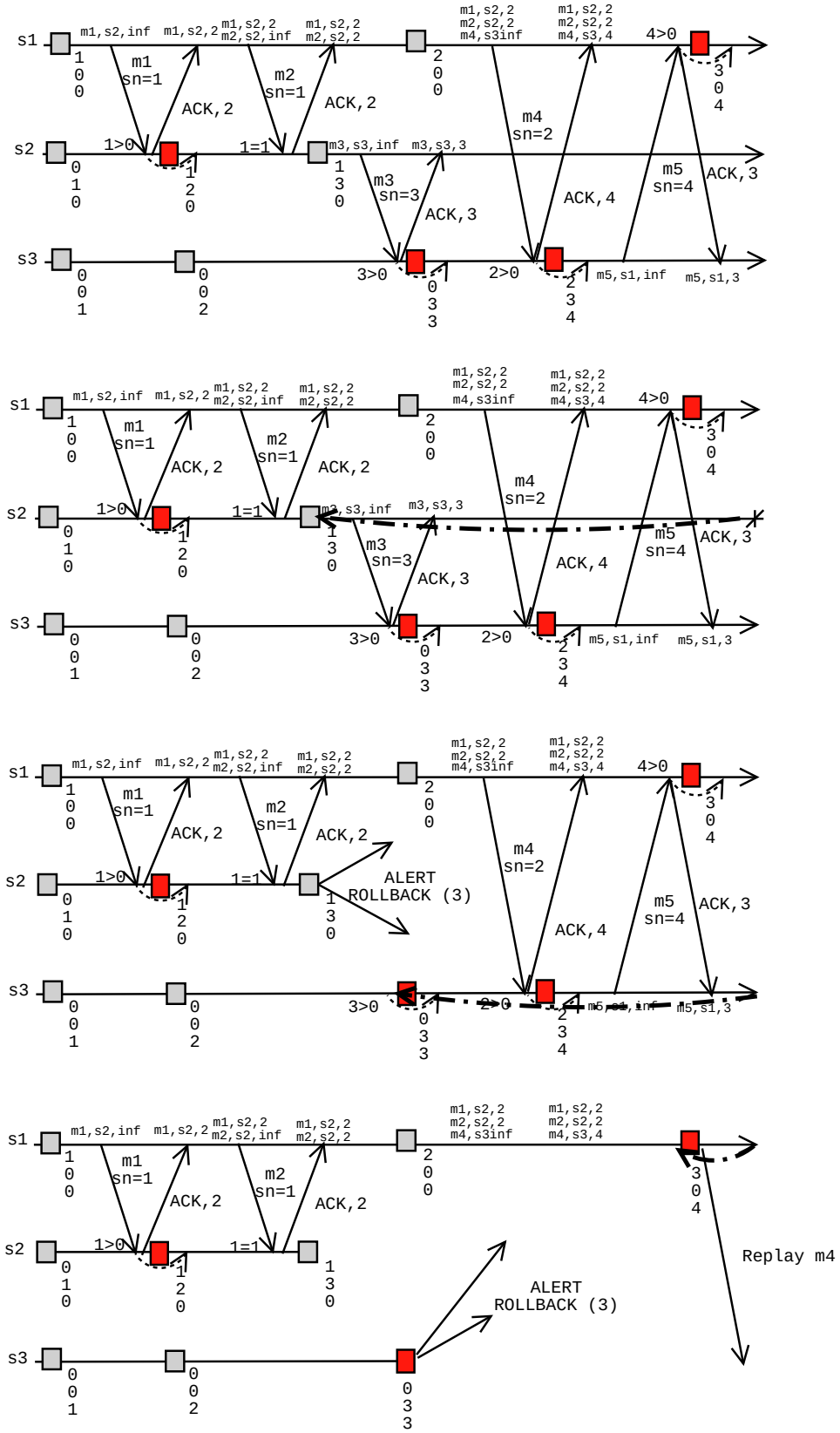

FIG. 3.13 – Retours arrières multiples

De nombreux points de reprise forcés Dans le cas où deux sites s'échangent des messages sous la forme de "questions-réponses", chaque message inter-site implique un point de reprise forcé.

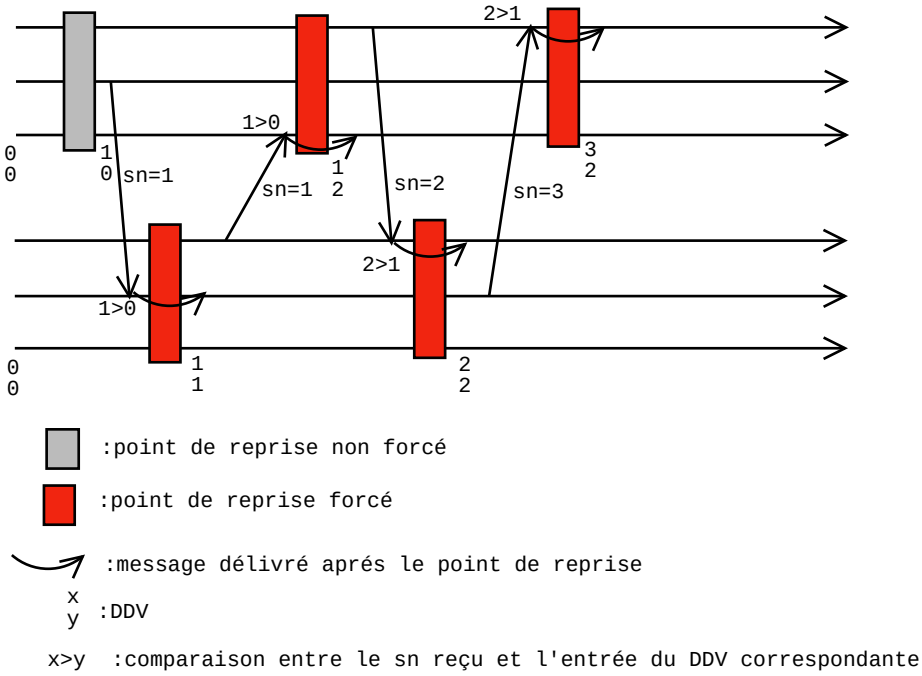

FIG. 3.14 – Trop de points de reprise forcés

Chapitre 4

Évaluation du protocole

L'implantation de mécanismes de tolérance aux défaillances demande une bonne connaissance de la programmation système, et beaucoup de temps.
La solution retenue pour évaluer les performances du protocole proposé a donc été la simulation à évènements discrets.
La première partie de ce chapitre présente le simulateur que nous avons créé afin d'évaluer le protocole proposé. La deuxième partie présente des résultats obtenus grâce à ce simulateur.

4.1 Présentation d'un simulateur à évènements discrets

4.1.1 Généralités

Dans un simulateur à évènements discrets, seuls les instants significatifs sont pris en compte. Il y a une double notion de temps : le temps physique, réel au cours duquel se déroule la simulation et le temps simulé. Les évènements sont associés à des dates et le temps de la simulation s'incrémente d'évènement en évènement.
La librairie utilisée pour le simulateur est la librairie C++SIM de l'université de Newcastle upon Tyne UK (http://cxxsim.ncl.ac.uk).
Cette librairie nous permet de définir des entités qui sont des classes C++ possédant une méthode *body()* dont le corp s'exécute dans un thread. Elle fournit également un ordonnanceur qui va gérer les file d'attentes des entités. Il est possible pour deux évènements distincts d'avoir une même date, cela permet donc de simuler des évènements parallèles sur un mono-processeur. En plus de l'ordonnanceur, il est possible d'interrompre explicitement l'exécution d'un thread pour donner le processeur à un autre puis de reprendre la main ensuite (ce procédé est beaucoup utilisé dans le simulateur) ce qui permet de communiquer entre les différents threads. La librairie C++SIM fournit également des classes de génération de flux de données aléatoires (selon différentes lois statistiques) et de statistiques (moyenne, écart type, histogrammes).

4.1.2 Description du simulateur

Le simulateur est composé de sept threads. Le thread main vérifie l'appel au programme (un argument de la ligne de commande doit donner le nom du fichier de configuration), charge la configuration de la simulation puis passe la main à un thread contrôleur, le thread contrôleur lance l'ordonnanceur et crée les quatres threads restants (qui sont en fait des entités au sens du simulateur C++SIM) composant le véritable simulateur. Les quatres entités sont les suivantes : *Nodes* qui représente l'ensemble des nœuds, *Network* qui représente le réseau, *Timers* pour simuler les timers et *Failure* pour les défaillances.
Le simulateur utilise des fichiers de configuration pour chaque simulation, ils sont au nombre de trois. Le premier décrit l'application, le deuxième décrit la topologie et enfin le troisième permet de régler les paramètres du protocole (essentiellement les durées des délais de garde).

L'application

L'application est décrite dans un fichier, elle doit correspondre à une certaine topologie de fédération de grappe (nombre de sites). En effet, il faut définir pour chaque site des temps d'initialisation et de calcul des nœuds (à l'aide de paramètres de lois statistiques), des probabilités d'émission de messages intra-site et vers chacun des autres sites, des probabilités sur les tailles de ces messages. Cela permet de représenter une grande variété d'applications.
Périodiquement, un nœud (le thread *Nodes* prenant l'identité d'un nœud) va invoquer la méthode *nextExecutionStatement(siteId)* en passant en paramètre son identifiant de site, la classe *Application* va lui retourner un patron de communication (sous forme d'une liste des communications à effectuer) ainsi qu'un temps de calcul, en respectant les paramètres de configuration de l'application.

La topologie

Le fichier de topologie décrit le nombre de sites, le nombre de nœuds dans chaque site et la demi-matrice des paramètres des liens réseaux. Cette demi-matrice (nombre de site x nombre de site) contient les latences et débits des liens entre chaque sites, ainsi qu'au sein des sites (les éléments de la diagonale). Il n'y a donc pas de distinction possible entre deux liens réseau au sein d'une même grappe.
Cette demi-matrice est utilisée par le réseau pour calculer les heures d'arrivée des messages.
La structure de donnée la plus couramment utilisée est un tableau de la taille du nombre de sites de la fédération, dons chaque entrée est un tableau du nombre de nœuds dans le site correspondant. Ce type de structure est utilisé pour les nœuds de la classe *Nodes*, les messages en attente dans le réseau de la classe *Network* et les heures de déclenchement des délais de garde de la classe *Timers*. La topologie contient donc un tableau (*nbNodesPerSite*) donnant les tailles permettant de parcourir ces structures.

Les Nœuds

L'entité *Nodes* représente tous les nœuds. Elle contient une structure de donnée "fédération" contenant un tableau (nombre de sites) de tableau de nœuds. Nous souhaitons pouvoir simuler des fédérations de grappes de calculateurs. Le nombre de nœuds à représenter peut donc être très élevé, il n'était donc pas envisageable de créer un thread par nœud. Lorsqu'elle est ordonnancée, cette entité prend l'identité d'un nœud particulier.
Le corps de ce thread est une boucle jusqu'à ce que le temps de l'application soit dépassé et tant qu'il reste des messages dans le réseau. A chaque itération l'entité est endormie pour un temps qui est calculé en fonction des temps de calcul de chaque nœud, puis prend l'identité du nœud correspondant, effectue un appel à *nextExecutionStatement(siteId)* de la classe "Application", effectue les communications retournées puis se repasse dans un état d'attente.
De plus ce thread peut recevoir des signaux. Une variable globale renseigne sur le type de signal reçu. Il peut s'agir d'un délai de garde, du réseau ou d'une faute. La réception de ces signaux engendre des appels aux fonctions correspondant aux algorithmes du chapitre précédent (*messageDispatching*, *timerHandler*...). C'est donc à ce niveau qu'est modélisé le protocole : à chacun des algorithmes présentés au chapitre 3 correspond une fonction C++.

Le réseau

Une entité a été créée pour représenter le réseau (*Network*). Le corps de ce thread est une boucle dans laquelle le thread est endormi pour un temps correspondant à la prochaine heure d'arrivée d'un message, puis place le message arrivé dans des variables globales avant de s'interrompre pour laisser la main au thread nœud (qui va traiter le signal en prenant comme identité celle du récepteur du message).
De plus, ce thread peut également traiter des signaux. En effet pour envoyer un message sur le réseau, le thread Nodes place le message dans une variable globale puis s'interrompt pour laisser le thread réseau traiter le message. Le traitement d'un message par le thread réseau correspond au calcul de son heure d'arrivée en fonction de la topologie et en son stockage dans la structure de donnée représentant le réseau. Ce message peut également être recopié par le réseau dans le cas d'une diffusion.

Délais de garde

Cette entité représente les différents délais de garde des différents nœuds. De même que l'entité *Network* il s'agit d'une boucle dans laquelle chaque itération commence par une attente (heure du prochain déclenchement du délai de garde), puis s'interrompt en laissant la main au thread *Nodes* après avoir placé le type de délai de garde et l'identité du nœud concerné dans des variables globales. Cette entité peut également recevoir des signaux du thread *Nodes* qui permettent de stopper, ou déclencher des délais de garde.

Faute

L'entité *Failure* est la plus simple. Elle attend un temps correspondant au MTBF de la fédération, suivant une loi exponetielle. En effet la loi exponentielle modélise bien les défaillances matérielles. Puis l'entité choisit un nœud selon une loi uniforme (le nœud défaillant pouvant n'importe lequel parmis les nœuds de la fédération), place son identité dans des variables globales et s'interrompt laissant la main au thread *Nodes*. Le nœud choisi n'est plus simulé, son absence est détectée par l'absence de message du type "je suis en vie", et le protocole de retour arrière est initialisé.
Il pourrait être intéressant de pouvoir choisir quel nœud est défaillant, à quelle heure.

4.2 Évaluation des performances par simulation

4.2.1 Validation du protocole et du simulateur

Dans un premier temps le simulateur a été utilisé pour vérifier le protocole. Il est compilable avec un niveau de trace élevé (cela se fait à la compilation pour des questions de performances), ce qui permet d'observer les échanges de messages, de contrôler les mises à jour des numéros de séquence, des vecteurs de dépendances directes, les acquittements et la recherche dans les journaux. Les délais de garde ont été réglés à des valeurs limites, ceci afin de tester la robustesse de l'algorithme (par exemple lorsque de nombreux nœuds initialisent un point de reprise en même temps) et du simulateur (si les messages entrent plus vite dans le réseau qu'ils n'en resortent, les "je suis en vie" suffisent à faire saturer le réseau).

4.2.2 Importance des paramètres

En faisant varier les paramètres de l'application (nombre, taille et nature des communications) et des délais de garde du protocole (temps entre deux points de reprise non forcés, entre deux déclenchements du ramasse miettes...) les résultats obtenus varient de manière très importante.

4.2.3 Influence des communications

Communications à double sens entre deux grappes Lorsque l'on a des grappes qui communiquent beaucoup entre elles et ce dans les deux sens, le nombre de points de reprise forcés est très grand (y compris si le nombre de points de reprise non forcés reste faible).
Dans l'exemple suivant, il y a deux sites de 50 nœuds. Les latences intra-grappe sont de l'ordre de 6.10^{-6} secondes et les débits sont de 60Mo/seconde. Entre les deux grappes, la latence est de 3.10^{-3} et le débit de 12Mo/secondes. Les communications inter-grappes sont importantes. L'application s'exécute pendant 2 heures, un point de reprise est déclenché tous les quarts d'heure dans chaque grappe, et le ramasse-miette est déclenché toutes les demi-heures.
Les fichiers de configurations commentés se trouvent en annexe du rapport. Les statistiques données

par le simulateur avec cette configuration sont les suivantes (les statistiques complètes sont fournies en annexe.) :

Pour le site 0 :	messages émis vers le site 1	4040
	messages reçus du site 1	2010
	points de reprise sauvegardés	1198
	non-forcés	1
	forcés	1197
	nombre maximum de points de reprise stockés	705
	après le ramasse miettes	2
	nombre maximum de messages stockés	51
	après le ramasse miettes	0

Pour le site 1 :	messages émis vers le site 0	2010
	messages reçus du site 0	4040
	points de reprise sauvegardés	1199
	non-forcés	1
	forcés	1198
	nombre maximum de points de reprise stockés	706
	après le ramasse miettes	2
	nombre maximum de messages stockés	26
	après le ramasse miettes	0

Les communications bidirectionnelles impliquent un nombre de points de reprise forcés trés élevé. Les sites prennent plus de 1000 points de reprise en 2 heures alors que le délai de garde ne se déclenche que tous les quart d'heures. D'après la trace, seul le premier point de reprise sur chacun des sites est pris grâce au déclenchement du délai de garde. Tous les autres sont des points de reprise forcés.
En plus de l'inconvénient du surcoût en terme de charge réseau, il faut également noter le coût en espace de stockage (le nombre de points de reprise stockés tient compte de la redondance nécessaire à l'implémentation de l'espace de stockage stable).

Communications à sens unique entre deux grappes Dès que les communications inter-grappe sont limitées, le nombre de points de reprise forcés l'est également (ainsi que les messages journalisés).
Un cas très favorable est celui où un site en "nourrit" un autre. En effet, le premier prend des points de reprise non forcés ce qui engendre des points de reprise forcés sur le deuxième (qui n'en prendra presque plus de non forcés) afin de limiter l'effet domino. Cet exemple reprend la même topologie que le précédent, le même fichier de configuration du protocole, et le même fichier de description d'application à la différence des probabilités d'émission inter-site. Ici il y a toujours un seul récepteur potentiel dans chacune des grappes, cependant, la probabilité d'envoyer un message de 0 vers 1 est de 0.8 au lieu de 0.5, et la probabilité d'envoyer un message de 1 vers 0 est de 0.0001. Les communications inter-grappes se font donc essentiellement dans le sens 0 vers 1.

Les statistiques données par le simulateur sont les suivantes :

Pour le site 0 :

messages émis vers le site 1	6351
messages reçus du site 1	1
points de reprise sauvegardés	8
non-forcés	7
forcés	1
nombre maximum de points de reprise stockés	6
après le ramasse miettes	2
nombre maximum de messages stockés	101
après le ramasse miettes	0

Pour le site 1 :

messages émis vers le site 0	1
messages reçus du site 0	6351
points de reprise sauvegardés	12
non-forcés	4
forcés	8
nombre maximum de points de reprise stockés	8
après le ramasse miettes	2
nombre maximum de messages stockés	1
après le ramasse miettes	0

Les résultats de cet exemple sont très différents de ceux du premier. Les nombres de points de reprise sauvegardés sont raisonnables (8 et 12), sur le site 1, qui reçoit pourtant plus de messages en provenance du site 0 que dans le premier exemple, il y a 8 points de reprises forcés et 4 déclenchés par délai de garde. En effet, le site 0 sauvegardant moins de points de reprise, il incrémente moins vite son numéro de séquence, le messages émis impliquent donc moins souvent un point de reprise.

Réglages des paramètres du protocole en fonction de l'application Pour chaque type d'application, il est intéressant de trouver un "réglage" des délais de garde offrant un bon rapport importance du retour arrière/surcoût engendré. Ainsi, pour une application ayant beaucoup de communications inter-grappe, les délais de garde déclenchant les points de reprise non forcés doivent être réglés à de grandes valeurs. En effet, le nombre de points de reprise forcés risque d'être suffisamment important.
On remarque que lorsque l'hypothèse que les communications inter-grappe sont très limitées est respectée, le protocole se comporte bien.

4.2.4 Le ramasse-miettes

La technique des points de reprise induits par les communications élimine bien l'effet domino, la ligne de recouvrement au pire se situe généralement peu en amont. Il n'est généralement nécessaire de conserver que un ou deux points de reprise par nœud (et donc les messages émis vers l'extérieur au cours d'une ou de deux périodes). Là aussi il faut trouver un bon compromis entre la fréquence des déclenchements du ramasse-miette (qui coûte cher en encombrement réseau) et le coût en espace

de stockage pour les points de reprise et les journaux.

Chapitre 5

Conclusion

Les fédérations de grappes de calculateurs sont des systèmes à très grande échelle, dans lesquels la probabilité d'apparition de défaillance est élevée. La mise en place de mécanismes de tolérance aux fautes dans ce type d'architecture est donc essentielle. Les protocoles de tolérance aux fautes pour les systèmes à grande échelle existants ne tiennent pas compte des particularités de l'architecture de fédération de grappes.
Le protocole proposé est un protocole hybride, utilisant aussi bien des mécanismes de points de reprise coordonnés que des mécanismes de points de reprise induits par les communications.
Le surcoût dû au protocole est difficile à évaluer. Il est nécessaire de trouver un bon compromis entre le nombre de points de reprise sauvegardés et l'ampleur du retour arrière en cas de faute. Un compromis doit également être trouvé entre l'encombrement en espace de stockage et le coût de lancement du ramasse miettes. Ces compromis dépendent de l'application, de la topologie et des performances du réseau.
De nombreuses améliorations peuvent être apportées au protocole proposé. La majorité des optimisations discutées au premier chapitre peuvent être intégrées à ce protocole : point de reprise en tâche de fond, point de reprise incrémental, et réduction du nombre de nœuds d'une grappe impliqués dans la sauvegarde d'un point de reprise (par la gestion des dépendances).
La notion de leader n'est pas indispensable: la détection de défaillance peut être répartie et pour tous les autres cas où on utilise un leader, un nœud de la grappe arbitraire peut donner les mêmes résultats qu'un leader fixé. La différence entre le numéro de séquence et le DDV peut être un paramètre. Dans le protocole étudié, une différence de 1 engendre un point de reprise forcé, il peut être envisageable de tolérer des différences plus grandes. Dans ce cas, le nombre de points de reprise forcés sera plus faible. En revanche une défaillance est susceptible d'engendrer un léger effet domino (qui est d'autant plus grand que la différence tolérée est importante). Ceci permettrait de traiter des applications communiquant intensivement, comme celle du premier exemple.
Le cœur de ce protocole, le fonctionnement inter-grappe, peut être utilisé pour des fédérations de grappes hétérogènes dans lesquelles certaines grappes utilisent une MVP et d'autres des systèmes de passage de messages. Il est également envisageable de l'implanter dans une fédération de grappes Kerrighed [42].

Bibliographie

[1] Adnan Agbaria and Roy Friedman. Starfish: Fault-Tolerant Dynamic MPI Programs on Clusters of Workstations. *The Eighth IEEE International Symposium on High Performance Distributed Computing*, pages 167–176, August 1999.

[2] Morin Christine Badrinath Ramamurthy. Common Mechanisms for Supporting Fault Tolerance in DSM and Message Passing Systems. Technical Report PI 1494, IRISA, November 2002.

[3] R. Baldoni, J. M. Hélary, A. Mostefaoui, and M. Raynal. Consistent State Restoration in Shared Memory Systems. In *Proc. of the Int. IEEE Conference on Advances in Parallel and Distributed Computing (APDC'97)*, pages 330–337, Shangai, 1997.

[4] A. Beguelin, E. Seligman, and P. Stephan. Application Level Fault Tolerance in Heterogeneous Networks of Workstations. Technical Report CMU-CS-96-157, School of Computer Science, Carnegie Mellon University, August 1996.

[5] George Bosilca, Aurelien Bouteiller, Franck Cappello, Samir Djailali, Gilles Fedak, Cecile Germain, Thomas Herault, Pierre Lemarinier, Oleg Lodygensky, Frederic Magniette, Vincent Neri, and Anton Selikhov. MPICH-V: Toward a Scalable Fault Tolerant MPI for Volatile Nodes, 2002.

[6] Kasidit Chanchio. *Efficient Checkpointing for Heterogeneous Collaborative Environments: Representation, Coordination, and Automation.* PhD thesis, Faculty of the Louisiana State University and Agricultural and Mechanical College, December 2000.

[7] K.M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. Computer Systems*, 3(1):63–75, February 1985.

[8] B. Chang, M. DeLap, J. Liszka, T. Murphy VII, K. Crary, R. Harper, and F. Pfenning. Towards a functional library for fault-tolerant grid computing. work-in-progress paper, 2002.

[9] Manuel Costa, Paulo Guedes, Manuel Sequeira, Nuno Neves, and Miguel Castro. Lightweight Logging for Lazy Release Consistent Distributed Shared Memory. In *Operating Systems Design and Implementation*, pages 59–73, October 1996.

[10] M. Singhal D. Manivannan. Asynchronous recovery without using vector timestamps. *Journal of Parallel and Distributed Computing*, 62:1695–1728, September 2002.

[11] M. Elnozahy, L. Alvisi, Y-M. Wang, and D.B. Johnson. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *ACM Computing Surveys (CSUR)*, 34:375–408, September 2002.

[12] Jean-Charles Fabre and Tanguy Perennou. A Metaobject Architecture for Fault-Tolerant Distributed Systems: The FRIENDS Approach. *IEEE Transactions on Computers*, 47(1):78–95, 1998.

[13] A. Ferrari. *Process State Capture and Recovery in HighPerformance Heterogeneous Distributed Computing Systems*. PhD thesis, Faculty of the School of Engineering and Applied Science at the University of Virginia, January 1998.

[14] Adam Ferrari. Process Introspection: A Checkpoint Mechanism for High Performance Heterogeneous Distributed Systems. Technical Report CS-96-15, Departement of Computer Science University of Virginia, Charlottesville VA 22903, 10, 1996.

[15] Pascal Fradet and Mario Sudholt. An Aspect Language for Robust Programming. In *ECOOP Workshops*, pages 291–292, june 1999.

[16] Felix C. Gartner. Fundamentals of Fault-Tolerant Distributed Computing in Asynchronous Environments. *ACM Computing Surveys*, 31(1):1–26, 1999.

[17] G. A. Geist, J. A. Kohl, and P.M. Papadopoulos. Providing Fault-Tolerance, Visualization an Steering of Parallel Applicaions. *International Journal of High Performance Computimg Applications*, 11(3):224–236, August 1997.

[18] Lord Hess. Preliminary design and development plan : Fault Tolerant Task. DataGrid, November 2001.

[19] J.M. Hélary, A. Mostefaoui, and M. Raynal. Communication-Induced Determination of Consistent Snapshots. *IEEE Transactions on Parallel and Distributed Systems*, 10(9):865–877, September 1999.

[20] Christine R. Hofmeister. Dynamic Reconfiguration of Distributed Applications. Technical Report CS-TR-3210, Departement of Computer Science University of Maryland, College Park MD 20742, 1994.

[21] Anne-Marie Kermarrec and Christine Morin. Smooth and Efficient Integration of High-Availability in a Parallel Single Level Store System. In *Proceedings of Euro-Par 2001*, page 30, August 2001.

[22] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[23] John C. Knight, Matthew C. Elder, and Xing Du. Error Recovery in Critical Infrastructure Systems. Technical Report CS-99-20, Departement of Computer Science University of Virginia, Charlottesville VA, April 1999.

[24] L. Lamport. Time, Clocks and Ordering of Events in Distributed Systems. *Communications of the ACM*, 21(7):558–565, July 1978.

[25] Renaud Lottiaux. *Gestion Globale de la Mémoire Physique d'une Grappe pour un Systèmes à Image Unique : mise en œuvre dans le système GOBELINS*. Thèse de doctorat, Université de Rennes I, December 2001.

[26] Pattie Maes. Concepts and Experiments in Computational Reflection. *ACM SIGPLAN Notices*, 22(12):147–155, December 1987.

[27] D. Manivannan and Mukesh Singhal. Quasi-Synchronous Checkpointing: Models, Characterization, and Classification. *IEEE Transaction on Parallel and Distributed Systems*, 10(7):703–713, July 1999.

[28] S. Monnet. Reprise d'Applications Parallèles dans les Grappes de Calculateurs, February 2003.

[29] C. Morin and I. Puaut. A Survey of Recoverable Distributed Shared Virtual Memory Systems. *IEEE Transaction on Parallel and Distributed Systems*, 8(9):959–969, September 1997.

[30] Christine Morin, Anne-Marie Kermarrec, Michel Banâtre, and A. Gefflaut. An Efficient and Scalable Approach for Implementing Fault Tolerant DSM Architectures. *IEEE Transactions on Computers*, 49(5):414–430, May 2000.

[31] R.H.B. Netzer and J. Xu. Necessary and sufficient Conditions for Consistent Global Snapshots. *IEEE Trans. Parallel and Distributed Systems*, 6(2):165–169, February 1995.

[32] Anh Nguyen-Tuong. *Integrating Fault-Tolerance Techniques in Grid Applications*. PhD thesis, Faculty of the School of Engineering and Applied Science at the University of Virginia, August 2000.

[33] H.S. Paul, A. Gupta, and R. Badrinath. Checkpoint and Recovery in Heteregeneous Cluster Based Systems, May 2002.

[34] H.S. Paul, A. Gupta, and R. Badrinath. Hierarchical Coordinated Checkpointing Protocol. In *International Conference on Parallel and Distributed Computing Systems*, pages 240–245, Novenber 2002.

[35] Hernani Pedroso. Fault Tolerant MPI for Clusters and Grid Environments. In *4th Plenary Workshop*, Italy, October 2001.

[36] J. Rough and A. Goscinski. Exploiting Operating System Services to Efficiently Checkpoint Parallel Applications in GENESIS. *Proceedings of the 5th IEEE International Conference on Algorithms and Architectures for Parallel Processing*, October 2002.

[37] Gregory T. Sullivan. Aspect-Oriented Programming using Reflection. In *OOPSLA*, October 2001.

[38] F. Sultan, T. D. Nguyen, and L. Iftode. Lazy Garbage Collection of Recovery State for Fault-Tolerant Distributed Shared Memory. *IEEE Transaction on Parallel and Distributed Systems*, 13(10):1085–1098, October 2002.

[39] Florin Sultan, Liviu Iftode, and Thu Nguyen. Scalable Fault-Tolerant Distributed Shared Memory. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing (CDROM)*, page 20. IEEE Computer Society Press, 2000.

[40] Tom Wagner and Don Towsley. *Getting Started With POSIX Threads*. Departement of Computer Science University of Masachusetts at Amberst, July 1995.

[41] Usenix Winter 1995 Technical Conference. *Libckpt: Transparent Checkpointing under Unix*, New Orleans LA, january 1995.

[42] G. Vallée, R. Lottiaux, L. Rilling, J.Y. Berthou, Ivan Dutka Malhen, and C. Morin. A Case for Single System Image Cluster Operating Systems: the Kerrighed Approach. *the Parallel Processing Letters journal*, 13(2), June 2003.

ANNEXES

Sébastien Monnet (Sebastien.Monnet@irisa.fr)

1 Premier exemple

1.1 Fichiers de configuration

1.1.1 Fichier de configuration de la topologie

```
2 // 2 sites
50 // de 50 noeuds chacun
50
0.000006 60000000 // latence et debit dans le site 0
0.003 12000000   0.000006 60000000 // latence et debit entre les 2 site
                                   // puis dans le site 1
```

Les latences sont en secondes et les debits en octets par seconde.

1.1.2 Fichier de configuration de l'application

Les durées sont en secondes, les deux valeurs correspondent aux bornes minimum et maximum de la loi uniforme :

```
7200 7200 // durée de l'application (2 heures)

// données pour le site 0
20 30 // temps d'imitialisation des noeuds dans le site 0
30 60 // temps de calcul des noeuds dans le site 0
0 // probabilité d'effectuer une diffusion
1024 10240 // taille du message a diffuser
2 // nombre de récepteurs potentiels dans le site
0.8 // probabilité d'envoyer au premier récepteur
1024 10240 // quantité à envoyer au premier récepteur
0.8 // probabilité d'envoyer au deuxième récepteur
```

```
1024 10240 // quantité à envoyer au deuxième récepteur
1 // nombre de récepteurs potentiels dans le site 1
0.5 // probabilité d'envoyer un message au recepteur du site 1
1024 10240 // quantité a envoyer

//données pour le site 1
10 15
60 120
0
1024 10240
1
0.9
1024 10240
1
0.5 // probablilité d'envoyer un message au site 0
1024 10240

// taille moyenne d'un état local sauvegardé
5000
```

Pour chaque site, le format de ce fichier est le suivant : temps de calcul, schéma de communication. Où le schéma de communication est représenté par un nombre de récepteurs potentiels (au sein du site, puis pour chacun des autres sites de la fédération de grappe), puis pour chacun des récepteurs éventuels, la probabilité d'émission, et la quantité de données à envoyer en cas d'émission.
Chaque nœud effectue un certain temps de calcul, communique, puis retourne dans une phase de calcul.

1.1.3 Le fichier de configuration des délais de garde du protocole (durées en secondes)

```
// pour le site 0
600 // verification des défaillances toutes les 10 minutes
120 // "je suis en vie" toute les minutes
900 // point de reprise tous les 1/4 heures
1800 // ramasse miette toutes les 1/2 heures
5 // aléat

// pour le site 1 (identique)
600
120
900
1800
6
```

1.2 Résultats

```
NETWORK TOTALS FOR SITE : 0
Intra-cluster messages (sent count) = 12702
Intra-cluster messages (rcv count) = 12702
Intra-cluster messages size (total) = 7.134e+07
Inter-cluster messages (sent count) = 4040
Inter-cluster messages (rcv count) = 2010
Inter-cluster messages size (total) = 2.26344e+07
I'm alive messages (count) = 5782
Request for checkpoint (count) = 58702
Acknowledgement for checkpoint (count) = 58702
Commit for checkpoint (count) = 58702
Checkpoint protocole messages size (total) = 176106
Request for stable storage (count) = 59900
Size (checkpoint sent) = 2.995e+08
Acknowledgement for stable storage (count) = 59900
Request for garbage collection (count) = 1
Acknowledgement for garbage collection (count) = 2
Commit for garbage collection (count) = 1
Collect for garbage collection (count) = 98
Garbage collection messages size (sent) = 862
Garbage collection messages size (rcv) = 1245
CPKT TOTALS FOR SITE : 0
Number of ckpts (commited) = 1198
Number of normals ckpts = 1
Number of forced ckpts = 1197
STORAGE TOTALS FOR SITE : 0
Maximum number of ckpt stored : 1410
Maximum number of ckpt stored after a garbage collection : 4
Maximum size of ckpt stored : 7.05e+06
Maximum number of messages stored : 51
Maximum number of messages stored after a garbage collection : 0
Maximum size of messages stored : 318075

NETWORK TOTALS FOR SITE : 1
Intra-cluster messages (sent count) = 3625
Intra-cluster messages (rcv count) = 3625
Intra-cluster messages size (total) = 2.02511e+07
Inter-cluster messages (sent count) = 2010
Inter-cluster messages (rcv count) = 4040
Inter-cluster messages size (total) = 1.1149e+07
I'm alive messages (count) = 5782
Request for checkpoint (count) = 58800
Acknowledgement for checkpoint (count) = 58751
Commit for checkpoint (count) = 58751
Checkpoint protocole messages size (total) = 176302
```

```
Request for stable storage (count) = 59950
Size (checkpoint sent) = 2.9975e+08
Acknowledgement for stable storage (count) = 59950
Request for garbage collection (count) = 2
Acknowledgement for garbage collection (count) = 1
Commit for garbage collection (count) = 1
Collect for garbage collection (count) = 98
Garbage collection messages size (sent) = 1245
Garbage collection messages size (rcv) = 862
CPKT TOTALS FOR SITE : 1
Number of ckpts (commited) = 1199
Number of normals ckpts = 1
Number of forced ckpts = 1199
STORAGE TOTALS FOR SITE : 1
Maximum number of ckpt stored : 1412
Maximum number of ckpt stored after a garbage collection : 4
Maximum size of ckpt stored : 7.06e+06
Maximum number of messages stored : 26
Maximum number of messages stored after a garbage collection : 0
Maximum size of messages stored : 175784
```

2 Deuxième exemple

2.1 Fichiers de configuration

Les fichiers de configuration de cet exemple sont identiques aux fichiers de configuration de l'exemple précedent, exepté le fichier de description de l'application qui devient:

```
7200 7200
20 30
30 60
0
1024 10240
2
0.8
1024 10240
0.8
1024 10240
1
0.8
1024 10240


10 15
```

```
60 120
0
1024 10240
1
0.9
1024 10240
1
0.0001
1024 10240

5000
```

2.2 Résultats

```
NETWORK TOTALS FOR SITE : 0
Intra-cluster messages (sent count) = 12702
Intra-cluster messages (rcv count) = 12702
Intra-cluster messages size (total) = 7.134e+07
Inter-cluster messages (sent count) = 6351
Inter-cluster messages (rcv count) = 1
Inter-cluster messages size (total) = 3.567e+07
I'm alive messages (count) = 5782
Request for checkpoint (count) = 392
Acknowledgement for checkpoint (count) = 392
Commit for checkpoint (count) = 392
Checkpoint protocole messages size (total) = 1176
Request for stable storage (count) = 400
Size (checkpoint sent) = 2e+06
Acknowledgement for stable storage (count) = 400
Request for garbage collection (count) = 1
Acknowledgement for garbage collection (count) = 2
Commit for garbage collection (count) = 1
Collect for garbage collection (count) = 98
Garbage collection messages size (sent) = 208
Garbage collection messages size (rcv) = 215
CPKT TOTALS FOR SITE : 0
Number of ckpts (commited) = 8
Number of normals ckpts = 7
Number of forced ckpts = 1
STORAGE TOTALS FOR SITE : 0
Maximum number of ckpt stored : 12
Maximum number of ckpt stored after a garbage collection : 4
Maximum size of ckpt stored : 60000
Maximum number of messages stored : 101
Maximum number of messages stored after a garbage collection : 0
```

```
Maximum size of messages stored : 591043

NETWORK TOTALS FOR SITE : 1
Intra-cluster messages (sent count) = 3625
Intra-cluster messages (rcv count) = 3625
Intra-cluster messages size (total) = 2.02511e+07
Inter-cluster messages (sent count) = 1
Inter-cluster messages (rcv count) = 6351
Inter-cluster messages size (total) = 8587.13
I'm alive messages (count) = 5782
Request for checkpoint (count) = 588
Acknowledgement for checkpoint (count) = 588
Commit for checkpoint (count) = 588
Checkpoint protocole messages size (total) = 1764
Request for stable storage (count) = 600
Size (checkpoint sent) = 3e+06
Acknowledgement for stable storage (count) = 600
Request for garbage collection (count) = 2
Acknowledgement for garbage collection (count) = 1
Commit for garbage collection (count) = 1
Collect for garbage collection (count) = 98
Garbage collection messages size (sent) = 215
Garbage collection messages size (rcv) = 208
CPKT TOTALS FOR SITE : 1
Number of ckpts (commited) = 12
Number of normals ckpts = 4
Number of forced ckpts = 8
STORAGE TOTALS FOR SITE : 1
Maximum number of ckpt stored : 16
Maximum number of ckpt stored after a garbage collection : 4
Maximum size of ckpt stored : 80000
Maximum number of messages stored : 1
Maximum number of messages stored after a garbage collection : 0
Maximum size of messages stored : 8587.13
```

3 Traces

Il est également possible de compiler le simulateur en définissant "TRACE1" et/ou "TRACE2" (selon le niveau de trace désiré), ce qui permet d'obtenir une trace complète de l'exécution. A chaque action (émission et réception de messages, déclechement d'un délai de garde, recherche dans le journal...) les nœuds indiquent l'heure de simulation et décrivent l'action courante. Cependant ces traces sont trés volumineuses et ne sont réellement exploitables qu'avec des outils comme *grep*.